

INTRODUCTORY PROBLEM SOLVING USING

ADA 95

SECOND EDITION

A. T. CHAMILLARD

REVERSE

INTRODUCTORY PROBLEM SOLVING USING ADA 95

SECOND EDITION

A.T. CHAMILLARD

Mc
Graw
Hill

The McGraw-Hill Companies, Inc.
Primis Custom Publishing

New York St. Louis San Francisco Auckland Bogotá
Caracas Lisbon London Madrid Mexico Milan Montreal
New Delhi Paris San Juan Singapore Sydney Tokyo Toronto

with draft
7

stronger mil. input
draft
208

drop in recruits
for mil.
86
before
beginning
83
have
to
fundraising
phil.
80

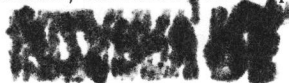
13

Re: The draft
ign
the answer
124

shoring up
voluntary
army
92

McGraw-Hill Higher Education 

A Division of The McGraw-Hill Companies



INTRODUCTORY PROBLEM SOLVING USING ADA 95

Copyright © 1999, 1998 by The McGraw-Hill Companies, Inc. All rights reserved. Printed in the United States of America. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base retrieval system, without prior written permission of the publisher.

McGraw-Hill's Primis Custom Publishing consists of products that are produced from camera-ready copy. Peer review, class testing, and accuracy are primarily the responsibility of the author(s).

1 2 3 4 5 6 7 8 9 0 CUS CUS 9 0 9

Part number ISBN 0-07-236818-7 of
set number ISBN 0-07-236972-8

Editor: M.A. Hollander

Cover Illustration: Chuck Rohrig

Cover Design: Grant Paul Design

Printer/Binder: Custom Printing

Dedication

This book is dedicated to my wife, Chris, and my children, Timothy, Nicholas, and Emily. Without your love and support, it never would have been completed. I love you all.

The book is also dedicated to all my Comp Sci 110 students, past, present, and future, at the U.S. Air Force Academy. Teaching you is an indescribable thrill; I hope this book helps make learning to program even more fun and rewarding.

procedure calc mi (time in float, air speed)
procedure calc km (miles in float, air speed)

Acknowledgments

I would like to acknowledge the reviewers of the draft manuscripts of this book, as well as the students and instructors who used and commented on the first edition. Their helpful, constructive comments helped strengthen the book considerably. Any remaining errors are mine alone.

Thanks are also due to Jason Moore, who helped me cut the CD included with the book.

Table of Contents

Chapter 1. Introduction.....	1
Chapter 2. Problem Solving.....	4
2.1. Polya.....	4
2.2. Wickelgren.....	6
2.3. Our Problem-Solving Steps	7
2.4. Conclusions.....	9
Chapter 3. Design.....	11
3.1. Sequential Control Structure	12
3.2. Selection Control Structure	14
3.3. Iteration Control Structure	16
3.4. Combining the Control Structures.....	18
3.5. Top-Down Decomposition	20
3.6. Conclusions.....	22
3.7. Exercises	22
Chapter 4. Testing.....	24
4.1. Testing Sequential Control Structures.....	24
4.2. Testing Selection Control Structures	26
4.3. Testing Iteration Control Structures	29
4.4. Testing Combinations of Control Structures	31
4.5. Completing the Test Plan.....	35
4.6. Conclusions.....	35
4.7. Exercises	36
Chapter 5. Your First Ada Program	38
5.1. What Does an Ada Program Look Like?.....	38
5.2. Comments	39
5.3. Using Other Packages.....	40
5.4. Identifiers.....	40

5.5. Variables and Constants.....	41
5.6. Simple Input and Output.....	41
5.7. A Matter of Style.....	43
5.8. Putting It All Together	43
Chapter 6. Data Types, Variables, and Constants.....	46
6.1. What's a Data Type?.....	46
6.2. Common Data Types.....	47
6.3. Choosing Between Variables and Constants.....	51
6.4. Giving Variables a Value	52
6.5. Input/Output of Different Data Types	54
6.6. Putting It All Together	60
6.7. Common Mistakes.....	64
6.8. Exercises	66
Chapter 7. Procedures	67
7.1. Deciding Which Procedures to Use.....	68
7.2. Figuring Out Information Flow	69
7.3. Creating the Procedure Header	70
7.4. The Rest of the Procedure	73
7.5. Calling the Procedure	76
7.6. Parameters and How They Work	80
7.7. Putting It All Together	81
7.8. Common Mistakes.....	90
7.9. Exercises	91
Chapter 8. Selection	92
8.1. If Statements	92
8.2. Case Statements	98
8.3. Putting It All Together	100
8.4. Common Mistakes.....	116
8.5. Exercises	116

Chapter 9. Iteration	117
9.1. For Loops	118
9.2. While Loops.....	122
9.3. Basic Loops	125
9.4. Putting It All Together	127
9.5. Common Mistakes.....	136
9.6. Exercises	137
 Chapter 10. Arrays	 138
10.1. Defining Array Types and Declaring Array Variables	138
10.2. Accessing Elements of the Array.....	140
10.3. Arrays and For Loops - "Walking the Array"	141
10.4. Multi-Dimensional Arrays.....	143
10.5. Putting It All Together	145
10.6. Common Mistakes.....	163
10.7. Exercises	163
 Chapter 11. File IO	 164
11.1. File Variables.....	164
11.2. Opening a File for Reading	165
11.3. Getting Input From a File	166
11.4. Closing the File When You're Done	167
11.5. End_of_File and End_of_Line	168
11.6. Creating a File for Writing	170
11.7. Printing Output to a File	171
11.8. Putting It All Together	171
11.9. Common Mistakes.....	188
11.10. Exercises	190
 Chapter 12. Putting It All Together	 191
12.1. A Tic-Tac-Toe Game	191

Chapter 13. Advanced Topics	223
13.1. Writing Your Own Packages	223
13.2. Enumeration Types.....	230
13.3. Using Records	233
13.4. Creating and Calling Functions	236
13.5. Overloading Procedures and Functions	237
13.6. Using Exception Handlers.....	240
 <i>Bibliography</i>	 243
<i>Index</i>	245

Chapter 1. Introduction

First and foremost, this is a book about problem solving. Our goal is to give you a process for solving the kinds of problems you'd expect to find in an introductory computer science course. One of the key things we'll talk about time and time again are *algorithms*. An algorithm is a detailed, step-by-step plan for solving a problem. Designing algorithms that are sufficiently detailed, and correct as well, is one of the hardest things to do when we solve a problem.

Unfortunately, it's hard to tell if an algorithm or set of algorithms really solves a given problem, so we go one step further with our solution by implementing our algorithms in computer programs written using Ada 95. As you might guess, Ada 95 has only been around for a few years, but its predecessor, Ada, has been in use for at least 15 years. In any case, this book is not designed as a complete reference for Ada 95 (though there are plenty of such references on the market). Instead, we'll focus on teaching you enough problem solving and Ada 95 (which we'll simply call Ada from now on) so you can solve introductory-level problems, without getting bogged down in Ada features that you'd never expect to use in an introductory course.

Chapter 2 presents some views about how people solve problems, then discusses the five-step process we'll use in this book to solve problems using Ada programs. The key idea behind the process is that the really hard work happens during the design step (which includes writing our algorithms), and that writing the Ada code given a detailed, correct algorithm is essentially a mechanical translation from one language to another. Inexperienced programmers have a hard time believing this, but learning the *syntax* (essentially the punctuation and grammar) of a programming language is not that difficult. The hard part comes from trying to get the *semantics* (really, the meaning of the program) right, and the meaning comes from the algorithms.

Chapter 3 is not actually Ada-specific, since it discusses the design process we use to develop our problem solutions (no matter which programming language we're going to use). This chapter introduces the

different *control structures* we use as building blocks in our problem solutions, and also covers *top-down decomposition*.

Chapter 4 is also not Ada-specific, since it discusses how we go about testing our programs once they've been written. We address the importance of testing, talk about how we decide which inputs to use when we design our test cases, and discuss how to complete the test plan when we actually test our programs.

In Chapter 5, we look at our first Ada program. We'll defer some of the details to later chapters, but we do discuss the importance of comments, how we can use code that other people have written, how we reserve memory in which to store things, and how we can do simple input to and output from those memory locations.

The question "What's a Data Type?" is asked (and answered) in Chapter 6. This is a very important concept in Ada, so you'll want to make sure you understand the idea and its implications. We also talk about how you decide between variables (things that can change values) and constants (things that don't change values), and show you how to give both of them values.

Chapter 7 discusses procedures and how we go about using them. Procedures tend to cause lots of difficulty for beginning (and even more experienced!) programmers, so we go through them slowly and carefully. We talk about how you decide which procedures to use, how you decide what information should flow in to and out of those procedures, and the mechanics of actually writing a procedure. Finally, we talk about how you actually tie those procedures together to implement your problem solution.

Chapter 8 talks about how we make decisions in our programs - how we select (or choose) between different alternatives. Since all but the most trivial programs have to make some choices, you should find that the constructs presented in this chapter come in handy when you're trying to solve problems.

The last basic control structure - iteration - is covered in Chapter 9. Iteration means we do something repeatedly, and we show you a number of ways we can iterate, or loop, in Ada. We show you how to do this when you know how many times to loop, and we also show you the loops to use when you're not sure how many times the loop will execute.

Chapter 10 covers arrays, which are particularly useful when we want to store lots of information but don't want to declare lots of variables. We show you how to define your very own data type for your arrays and how to declare variables of that type. We also show you how to get at the elements (the parts) of an array, and the most common way to "walk the array" to process the information it contains.

Chapter 11 talks about how we can save and retrieve information from files on our disks. This is particularly useful if you want to save the information for later use. We show you how to open existing files and read from them, how to create a new file and write to it, and how to close the files when you're done. You'll find that files are a lot like books - to read one, you have to open it first, and when you're done you should close it so you don't break the binding!

Chapter 12 uses the information from the earlier chapters to solve a number of relatively large problems. It's important to realize that the techniques and constructs we discuss in those chapters can be applied to larger problems, so we carefully walk through the solutions of a few problems that could actually be encountered in real life.

The final chapter (Chapter 13) covers topics that aren't necessary for an introductory course, but allow some exploration of more advanced Ada topics. We show you how to write your own packages and functions, discuss exceptions and how to handle them, and cover some additional topics on renaming and overloading functions and procedures.

Most chapters contain a section on common mistakes and a set of exercises. We're sure we haven't described every possible mistake a beginning programmer might make, but we hope the common mistakes sections serve as a useful starting point as you try to debug your programs. The exercises give you an opportunity to hone your problem solving and Ada programming skills, and also provide your teacher with a resource from which to assign you problems!

So there you have it - this is a book about problem solving, using Ada to implement those problem solutions. Let's get to it!

Chapter 2. Problem Solving

Programming is really all about problem solving -- given some list of requirements for a problem, you need to generate a solution that meets those requirements. In this book, we'll have you implement your solutions as Ada programs, both because you can then easily check your solution for correctness and because we believe that implementing your solution helps you solidify your understanding of the problem-solving process. Although it might seem like this is a book about Ada programming, it's really a book about solving problems using Ada. In fact, the problem-solving techniques you learn in this book can be applied any time you need to solve a problem, even if you're not going to use a computer to do it.

So the big question is "How do people solve problems?" Many researchers have tried to answer this question, but we'll only consider two of them in this chapter -- Polya and Wickelgren. We'll then develop a series of problem-solving steps (heavily based on Polya's work) that we'll use throughout the rest of the book, and which you can use whenever you're faced with a problem to be solved.

2.1. Polya

Many years ago, George Polya wrote a book called *How to Solve It* [Pol54], in which he set forth a series of steps people can use to solve problems. Although his focus was on mathematical problems, the steps are sufficiently general to apply to all problems.

Polya's Problem-Solving Steps

1. Understanding the Problem
2. Devising a Plan
3. Carrying Out the Plan
4. Looking Back

Understanding the Problem

The first step in Polya's system is to understand the problem. This certainly seems like common sense -- how can you possibly solve the problem correctly if you don't know what it is? Surprisingly, lots of people try to jump right into finding a solution before they even know what problem they're trying to solve. This may be due, at least in part, to the feeling that we're not making progress toward the solution if we're "just" thinking about the problem. In any case, you need to make sure you understand the problem before moving on.

Devising a Plan

The second step is to try to come up with a *high-level* plan for solving the problem. This plan doesn't worry about all the details, but it will be used to guide our solution in the next step. For example, say you need to find the length of the hypotenuse of a right triangle. In this step, we'd simply decide that our plan is to use the Pythagorean Theorem, and we defer the details until later. This step is important because without a high-level plan we can get mired down in the details of our solution without making progress toward our true goal.

Carrying Out the Plan

There's a common saying that "the devil is in the details". You may come up with a great-looking high-level plan, but in this step you actually have to carry out the plan. In other words, here's where you actually have to work out the details. Sometimes that's not too bad -- in our hypotenuse example, we'd simply apply the Pythagorean Theorem by taking the square root of the sum of the squares of the two "legs" of the triangle. In other cases, however, the details may be so difficult to work out that we need to revisit the previous step to rethink our high-level plan. When this third step of Polya's system is complete, though, we have a detailed, step-by-step solution to the problem.

Looking Back

Polya's last step is to go back and look at our problem solution and the process we used to find it. Does the solution actually solve the problem we were given? Could we have found the solution another way? Are there

other, similar problems to which our solution process could also be applied? This step is important since it confirms that we've actually solved the problem and also helps us "store" our solution (and process) for later use on other problems.

2.2. Wickelgren

Polya's system is largely concerned with the *process* of synthesizing our solution. In contrast, Wayne Wickelgren's system [Wic79] is more concerned with *analysis* of the problem at hand. Like Polya's system, Wickelgren's system has four components, but as you'll see, they reflect a different approach to problem solving.

Wickelgren's Components

1. Givens
2. Operations
3. Goal
4. Solution

Givens

The givens are the "things that are true" in our problem domain. For example, say you're trying to solve the Rubik's Cube puzzle. The givens would be facts like "there are 9 green squares", "there are 9 red squares", and so on. The givens provide a starting point for our problem solution.

Operations

The operations are "things we can do" in our problem domain. Continuing with our Rubik's Cube example, valid operations include things like "rotate the bottom row 90 degrees to the right", "rotate the front face 90 degrees clockwise", and so on.

Goal

The goal is the "place" we're trying to get to in the problem domain. For the Rubik's Cube, our goal is to get one face of the cube to be green, one to be red, etc. For other problems, our goal could be to process numbers

in certain ways, have \$1 million in the bank, get As in all our courses, or to break 12 hours in an Ironman.

Solution

Wickelgren's solution consists of the set of givens, the set of operations, the goal, and the sequence of operations we need to apply (the steps we need to take) to get from the starting point (the original givens) to the goal. The real trick is to figure out the correct sequence of steps. There is, of course, no perfect way to come up with that sequence -- that's why problem solving is hard in the first place! Wickelgren does provide some suggestions for coming up with the sequence of steps, though; breaking the goal into subgoals, working backward from the goal for certain problems, using solutions we've developed in the past, and so on.

2.3. Our Problem-Solving Steps

Neither of the problem-solving systems we've discussed exactly matches what we'd like to use in this book, though Polya's comes pretty close. Here are the problem-solving steps we'll use in this book:

Our Problem-Solving Steps

1. Understand the Problem
2. Design a Solution
3. Write a Test Plan
4. Write the Code
5. Test the Code

Understand the Problem

This was a critical step in Polya's system for good reason. Understanding the problem requires that you understand **WHAT** is required. One researcher found that "successful problem solvers spend two to three times longer reading an initial problem statement than did unsuccessful problem solvers" [Woo94]. The successful problem solvers were making sure they understood what the problem was before rushing off to design a solution.

If you want to successfully solve the problems in this book (and the problems your teacher gives to you!), you need to do the same thing.

As an example, consider the following problem: "Look through a deck of cards to find the Queen of Hearts." Do you REALLY understand the problem? Do you know what to do if you're not playing with a full deck <grin>? Do you know what to do if the Queen of Hearts isn't in the deck? Do you know what to do after you find the Queen of Hearts? Even the simplest problems require you to think carefully before proceeding to the next problem-solving step.

Design a Solution

Once you know what the problem is, you need to formulate a solution. In other words, you need to figure out **HOW** you're going to solve the problem.

Designing your solution is, without a doubt, the hardest problem-solving step. That's why we'll dedicate Chapter 3 to a discussion of how we go about doing our design. For now, let's just say that completing this step results in an algorithm (or a set of algorithms) designed to solve the problem.

Write a Test Plan

Our third problem-solving step helps you make sure you've actually solved the problem. Does your solution do what it's supposed to? It's reasonable to go back and run your program (which you'll write in the next step) a few times to make sure it solves the right problem. You're not really worried about making your program more efficient here, you just want to make sure it's correct.

But how can you decide how to test your program if you haven't even written it yet? We'll talk a lot more about software testing in Chapter 4, but it turns out that you can decide how to test your program just by knowing the requirements and the program structure (which you can get from your algorithms). Basically, writing a test plan requires that you write down how you're going to test your program (which inputs you're going to use) and what you expect to happen when you use those inputs.

Write the Code

This step is where you take your detailed algorithms and turn them into a working Ada program (computer programs are commonly called *code*). Believe it or not, whether you've ever written a program in your life or not, this is the EASIEST step of all. Certainly, you'll have to learn the required syntax for Ada (just as you have to learn the grammar rules for a foreign language you try to learn), but syntax is easy. If you've done a good job determining how to solve the problem in your algorithms, writing the code will be a snap. As a matter of fact, Nico Lomuto says "Discovering a solution to a programming problem is a complex, iterative process in which devising the *logic* of the solution is usually a far more demanding task than expressing that logic in a particular programming language" [Lom87]. One more time -- the hard part in the process is designing your solution, not writing the code.

Test the Code

Finally, the last step in our problem-solving process. Now that you've implemented your solution, you use your test plan to make sure the program actually behaves the way you expected it to (and the way it's supposed to). Again, more about this in Chapter 4.

2.4. Conclusions

Problem solving is an iterative process no matter what set of steps we use. For example, we may get all the way to testing our code before realizing that we didn't design a correct solution, so we'd need to go back to that step and work our way through to the end again. Similarly, we may realize as we write our code that we forgot something in our test plan, so we'd return to that step to make our corrections. There's nothing wrong with going back to previous steps; as a matter of fact, it's a natural part of the problem-solving process, and you'll almost always need to iterate a few times before reaching a correct problem solution.

You should also know that research has shown that "good problem solvers are more aggressive, confident, tenacious, and attentive to detail. They placed their faith in reason, rather than in guessing. In contrast, the efforts of poor problem solvers were characterized by a lack of attention to

detail, unreasoned guessing, and self-justification" [Woo94]. Problem solving requires a careful, step-by-step progression to a solution, not random guessing or waiting for bursts of inspiration. If you carefully follow the problem-solving steps listed above, going back to previous steps as necessary, we're sure you'll be able to successfully solve the problems in this book (and other assigned problems). Even more importantly, you can use these steps whenever you're faced with a problem, whether or not you're going to use a computer to solve it.

Chapter 3. Design

For all but the simplest problems, designing your solution is done in two steps.

In the first step, you break the problem down into smaller subproblems, which will be easier to solve than the big problem. This is commonly called *top-down decomposition* because we're decomposing (breaking down into smaller pieces) our problem. We'll work through an example of top-down decomposition later in the chapter.

In the second step, you write an *algorithm* for each subproblem. Recall that we defined an algorithm as "a step-by-step plan for solving a problem." These algorithms need to be very detailed -- if you handed the algorithm to your crazy Uncle Rick, could he follow it correctly? One good technique for writing correct, detailed algorithms is to think of other, similar problems you've already solved and re-use parts of those solutions. As you solve more and more problems, you'll develop a large collection of subproblem solutions that you can borrow from as appropriate.

OK, let's go back to our example problem from Chapter 2 (looking for the Queen of Hearts). This problem is small enough that we don't really need to break it into subproblems, so let's just try to write an algorithm to solve the problem. Your first try might look something like this:

- Look through the deck
- If you find the Queen of Hearts, say so

What do you think? Is this a good algorithm? Stop nodding your head yes! This is not nearly detailed enough to be useful. Your crazy Uncle Rick has no idea how to "Look through the deck." Let's try to make the algorithm more detailed:

- If the top card is the Queen of Hearts, say so
- Otherwise, if the second card is the Queen of Hearts, say so
- Otherwise, if the third card is the Queen of Hearts, say so

...

Well, this is better, but now it looks like our algorithm will be REALLY long (52 steps for a deck of 52 cards). Also, our algorithm has to know exactly how many cards are in the deck. However, we can look at our algorithm and see that we're doing the "same" thing (looking at a card) many times. Because we're doing this repeatedly, we can come up with a much cleaner algorithm:

- While there are more cards in the deck
 - If the top card is the Queen of Hearts, say so and discard the Queen
 - Otherwise, discard the top card

We now have a correct, detailed algorithm that even Uncle Rick can follow, and it doesn't make any assumptions about the number of cards in the deck.

When we write our algorithms, we use three types of *control structures* - *sequential*, *selection*, and *iteration*. Sequential simply means that steps in the solution are executed in sequence (e.g., one after the other, in order). Selection means that steps in the solution are executed based on some selection, or choice. Iteration means that certain steps in the solution can be executed more than once (iteratively). We'll look at these control structures in more detail in the following sections, but you should understand that we simply use these structures as building blocks to develop our algorithms.

3.1. Sequential Control Structure

The simplest problem solutions just start at the beginning and go to the end, one step at a time. Let's take a look at an example:

Example 3.1 Reading in and Printing Out a Band Name

Problem Description: Write an algorithm that will read in the name of a band, then print that name out.

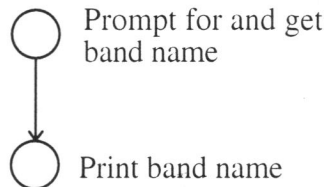
Algorithm: Here's one way we could solve this problem:

```
-- Prompt for and read in band name
-- Print out band name
```

Note that we have to make sure we ask for the band name before reading it in; if you walked up to someone and simply stared at them, would they know you were waiting for them to give you the name of a band?

For many people, a "picture" of their solution helps clarify the steps and may also help them find errors in their solution. We would therefore also like a way to represent our algorithms pictorially. One popular way to pictorially represent problem solutions is by using flowcharts, which show how the solution "flows" from one step to the next. We'll use the same idea in something called a *Control Flow Graph (CFG)*, which captures the same information as a flowchart without requiring a bunch of special symbols. A CFG graphically captures the way control flows through an algorithm (hence the name).

Let's look at the CFG for our algorithm. The CFG looks like:



Because we have a sequential control structure (the program simply does one step after another), we have a sequence of *nodes* in the CFG. Each node represents a step in our algorithm; the *edges* in the CFG represent ways we can get from one node to another. In the algorithm above, we only have one way to get from the first step to the last step (since this is a sequential algorithm), so we simply add an edge from the first node to the last node.

The sequential control structure gives us our first building block for creating our problem solutions, but many of the problems we try to solve will also require us to make decisions in the problem solution. That's where the next control structure comes into play.

3.2. Selection Control Structure

Suppose we need to make a decision in our problem solution? The sequential control structure won't be sufficient, because it doesn't let us select between different courses of action. Not surprisingly, the selection control structure is designed to let us do exactly that. Let's look at an example:

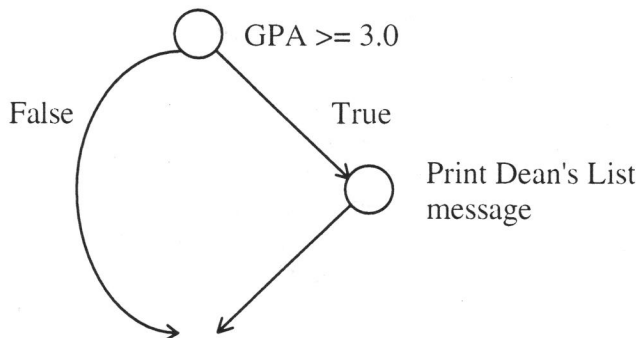
Example 3.2 Printing a Dean's List Message

Problem Description: Write an algorithm that will print a Dean's List message if the student's GPA is 3.0 or above.

Algorithm: Here's one solution:

```
-- If GPA greater than or equal to 3.0
-- Print Dean's List message
```

Basically, our solution will print out the Dean's List message if the GPA is 3.0 or higher; otherwise, it won't do anything. Notice our use of indentation in the algorithm. We indent the second step to show that the second step is only accomplished if the GPA is 3.0 or higher. The associated CFG looks like this:



When we use a selection control structure, we end up with two or more *branches*. If the GPA is greater than or equal to 3.0, we would take the branch on the right (the True next to the branch means that the statement

"GPA >= 3.0" is true) and print the Dean's List message. If the GPA is less than 3.0, we take the branch on the left. Because this branch doesn't have any nodes, the solution will continue on to the next step without printing (or doing) anything on that branch.

Labeling the branches True and False may seem a little awkward to you right now (Yes and No might seem to be more intuitive labels, for example). The reason we chose the above labels is because "GPA >= 3.0" is called a *boolean expression*. A boolean expression is an expression that evaluates to one of two values -- true or false. For example, the boolean expression GPA >= 3.0 evaluates to true if the value of GPA is greater than or equal to 3.0; otherwise, it evaluates to false.

Notice that both arrows at the bottom of the above CFG "end in thin air". Remember, these control structures are building blocks; if our algorithm had more steps after the selection, we would simply plug in the appropriate CFG in the space after the selection portion of the CFG. We'll show an example of this later in this chapter.

For now, let's make our Dean's List example a bit more complicated:

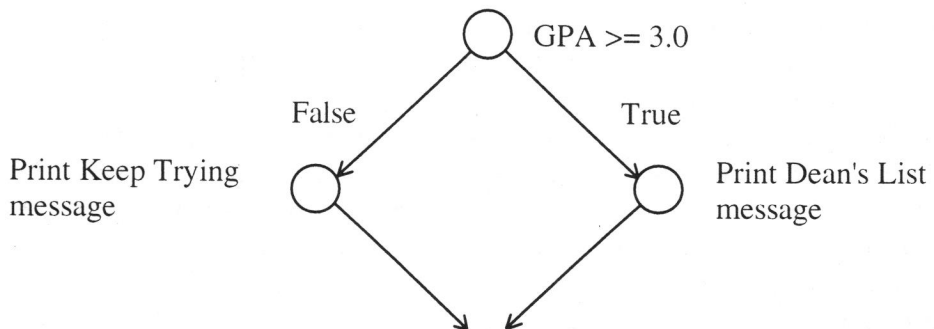
Example 3.3 Printing a Dean's List or Keep Trying Message

Problem Description: Write an algorithm that will print a Dean's List message if the student's GPA is 3.0 or above, and will print a Keep Trying message if it's not.

Algorithm: Our solution builds on the one above:

```
-- If GPA greater than or equal to 3.0
  -- Print Dean's List message
-- Otherwise
  -- Print Keep Trying message
```

Basically, our solution will print out the Dean's List message if the GPA is 3.0 or higher; otherwise, it will print out the Keep Trying message. We again use indentation in the algorithm, this time to show that the second step is only accomplished if the GPA is 3.0 or higher and that the fourth step is only accomplished if the GPA is less than 3.0. The associated CFG looks like this:



The branch on the right behaves exactly as it did in our previous solution, but this time the branch on the left (which we follow when GPA is less than 3.0) prints a Keep Trying message.

So now we have control structures that will simply do one step after another (sequential) and that will let us select between different branches (selection). All we need is one more control structure, and we'll have all the building blocks we need to solve ANY problem.

3.3. Iteration Control Structure

What's the last thing we really need in our problem solutions? The ability to accomplish certain steps in the algorithm multiple times. In other words, we'd like to iterate (repeat) those steps in the algorithm. That's where the third and final control structure comes in - the iteration control structure. Iteration control structures are often called *loops*, because we can loop back to repeat steps in our algorithm. Let's look at an example:

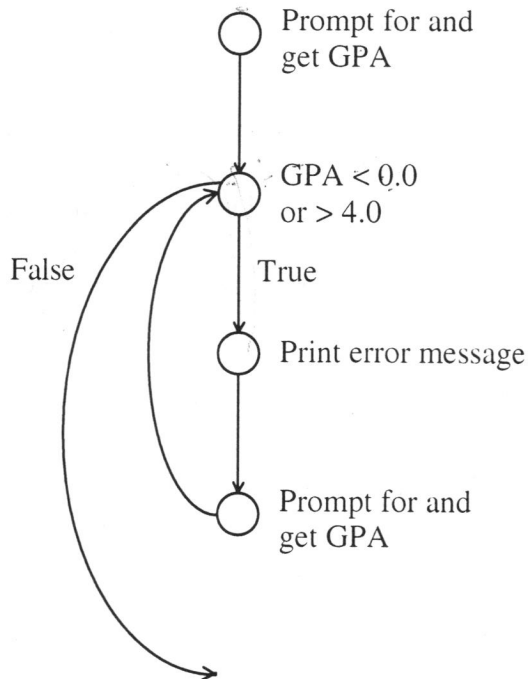
Example 3.4 Getting a Valid GPA

Problem Description: Write an algorithm that will ask for a GPA until a valid GPA is entered.

Algorithm: Here's one solution:

```
-- Prompt for and get GPA
-- While GPA is less than 0.0 or greater than 4.0
  -- Print error message
  -- Prompt for and get GPA
```

Basically, our solution asks for a GPA. While the GPA is invalid (less than 0.0 or greater than 4.0), we print an error message and ask for a new GPA. Eventually, the user will enter a valid GPA, stopping the iteration of the last three steps in the algorithm. This time, we use indentation to show that the third and fourth steps are contained inside the loop (we call these steps the *loop body*). The associated CFG is on the following page.



The first thing we do is prompt for and get a GPA from the user. We then move to the node that "tests" to see if the GPA is invalid. Note that the test in this node is a boolean expression like the one we saw for the selection control structure. If the GPA is invalid, we take the branch marked True, which lets us print an error message and read in a new GPA. Notice the edge from the bottom node back up to the "test" node; after

we've read in the new GPA, we go back to the "test" node to check if the new GPA is invalid. Eventually, the user will enter a valid GPA, the boolean expression at the "test" node will evaluate to False, and we'll take the left branch (the one marked False) out of the iteration control structure.

So that's it! We have all the control structures we need to solve ANY problem. We still have to decide how to connect our building blocks into a reasonable problem solution, of course, but at least we know we have all the blocks we need.

3.4. Combining the Control Structures

Let's combine the control structures to solve an even more complicated problem. Here it is:

Example 3.5 Getting a Valid GPA and Printing a Message

Problem Description: Write an algorithm that will ask for a GPA until a valid GPA is entered, then print a Dean's List message if the GPA is greater than or equal to 3.0 or print a Keep Trying message if the GPA is less than 3.0.

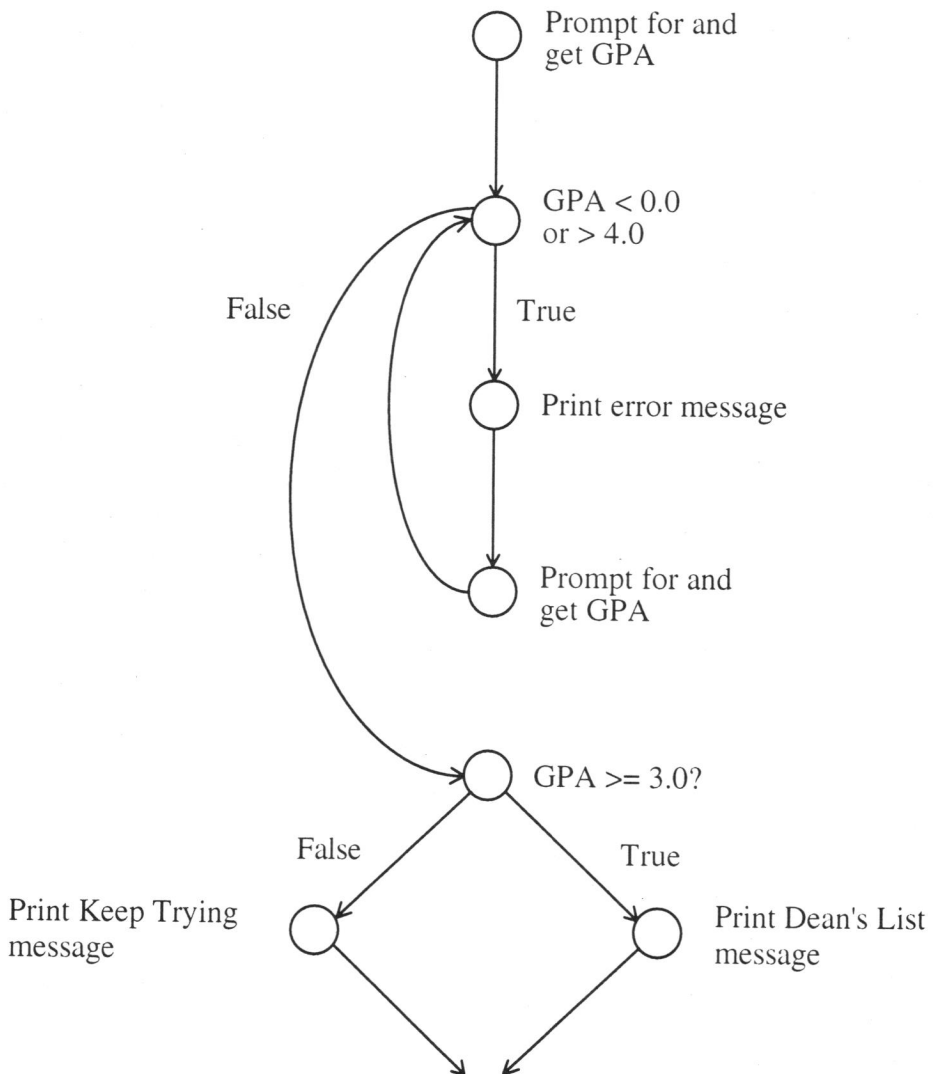
Algorithm: Here's one solution:

```
-- Prompt for and get GPA
-- While GPA is less than 0.0 or greater than 4.0
  -- Print error message
  -- Prompt for and get GPA
-- If GPA greater than or equal to 3.0
  -- Print Dean's List message
-- Otherwise
  -- Print Keep Trying message
```

This is really just a combination of the solutions we did earlier, but that's exactly how people solve problems! We look at a new problem, then go back and see if we've solved similar problems in the past. Even if we've never solved the exact same problem, we've probably solved a problem (or at least parts of the problem) that's similar. We then use those "old"

solutions to build our solution to the problem at hand. The associated CFG is on the following page.

We can combine the three control structures as much as we want, connecting those building blocks together to construct our problem solutions.



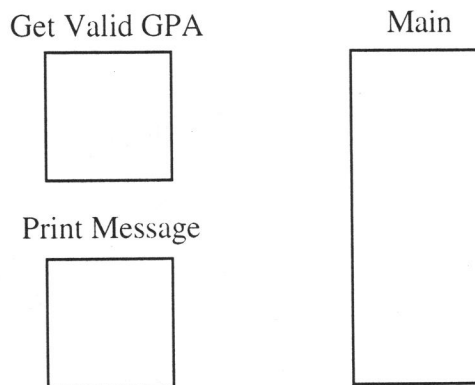
3.5. Top-Down Decomposition

When we try to solve larger problems, we find that solving the problem "all at once" becomes very difficult. Coming up with a problem solution is easier if we use top-down decomposition, where we break the problem into smaller subproblems, solve each of the subproblems, then combine them into a solution to the large problem.

Let's use top-down decomposition on the problem in Example 3.5. One effective technique to use for deciding on the subproblems is to write a high-level algorithm for the problem solution, like:

```
-- Get a valid GPA
-- Print the appropriate message
```

Note that this algorithm is NOT detailed enough for crazy Uncle Rick to follow; the detail comes in the solutions to each of the subproblems. It seems clear that, for this example, we should have one subproblem that gets a valid GPA and another that prints an appropriate message for that GPA. Let's start drawing a *decomposition diagram*, which is simply a picture of the subproblems and the "main" solution (the high-level algorithm above):



Now that we've decided what our subproblems are, we need to decide what information flows between those subproblems and our main solution.

For example, after the Get Valid GPA subproblem gets a valid GPA, what should it do with that GPA? It should send it out to the main solution. Similarly, the Print Message subproblem needs to know what the GPA is to print an appropriate message, so the main solution needs to send the GPA in to the Print Message subproblem. We therefore also include the information flow in our decomposition diagram (see the diagram on the following page)

We use the arrows between the subproblems and the main solution to show the direction of the information flow, and we label the arrows to indicate what information is flowing.

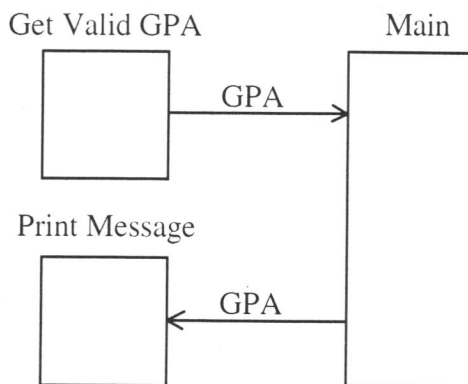
To complete our problem solution, we'd write detailed algorithms for our two subproblems (we can reuse our work from Example 3.5):

Get Valid GPA

```
-- Prompt for and get GPA
-- While GPA is less than 0.0 or greater than 4.0
  -- Print error message
  -- Prompt for and get GPA
```

Print Message

```
-- If GPA greater than or equal to 3.0
  -- Print Dean's List message
-- Otherwise
  -- Print Keep Trying message
```



Now, what would happen if we changed our high-level algorithm to do all this twice:


```
-- Get a valid GPA
-- Print the appropriate message
-- Get a valid GPA
-- Print the appropriate message
```

Would our decomposition diagram change? Not at all! Even though we'll use each of our subproblem solutions twice, we only need to show them once in the diagram. Why is that? Because once we've solved each of the subproblems, we can reuse those solutions as many times as we want "for free." And our algorithms for those subproblems wouldn't change, either; just because we USE an algorithm twice doesn't mean we have to WRITE it twice.

That's all there is to top-down decomposition. We'll practice this a lot more in later chapters.

3.6. Conclusions

Designing your solution to a problem is the hardest step in the problem-solving process. For most problems, you'll need to use top-down decomposition to decide how to break the problem into subproblems and how information flows between those subproblems and your main solution. You'll also need to write detailed, step-by-step algorithms for each of those subproblems (and a higher-level algorithm for your main solution). It seems like a lot of work, and it is, but taking the time to do good, detailed designs will help you successfully complete the remaining problem-solving steps as you generate your problem solutions.

3.7. Exercises

1. Design a solution to read in the X, Y, and Z coordinates for 2 points, calculate the distance between them, and display the distance.
2. Design a solution to read in two GPAs and print which GPA is higher.

3. Design a solution to read in three numbers from the user and print them in ascending order.
4. Design a solution to find and print the highest number in a set of 10 numbers.
5. Design a solution to search through a list of numbers and determine if the number 1983 is in the list.

Chapter 4. Testing

Suppose you're given a problem description and a computer program that supposedly solves the problem -- how do you know that it really does? Or say you've solved a problem in your class by writing an Ada program. You'd like to make sure it actually works before turning it in for a grade, wouldn't you? How do you do that?

One way to try to make sure a program does what it's supposed to do is with *testing*; which is running a program with a set of selected inputs and making sure the program gives the correct outputs. Choosing which inputs to use is the trick, of course, because most programs can accept a huge number of inputs; if we use all of them, testing will take a LONG time! So how do we pick which inputs to use? We'll cover that in more detail soon, but at a high level we can either use the list of requirements (the things our solution is supposed to do) to pick our inputs (typically called *black box testing*) or we can use the structure of our solution to pick our inputs (typically called *white box testing*). Although both techniques can be effective, in this book we'll use the structure of our solution (e.g., the algorithms and CFGs) to pick our inputs.

In this chapter, we talk a lot about how we go about testing a program. You should understand that we could use the same techniques to test our algorithms, though. As a matter of fact, you'll see that we decide how to test based on our algorithms and CFGs, then do the actual testing on our program.

Since we're going to use program structure as the basis of our testing, let's take a look at how we go about testing each of the control structures we discussed in the last chapter.

4.1. Testing Sequential Control Structures

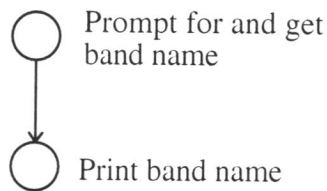
Let's revisit our sequential control structure example from the last chapter:

Example 4.1 Reading in and Printing Out a Band Name

Problem Description: Write an algorithm that will read in the name of a band, then print that name out.

Algorithm and CFG:

```
-- Prompt for and read in band name
-- Print out band name
```



Testing this program will be pretty easy -- all we have to do is run it once to make sure it prompts for, gets, and correctly prints out a band name. We simply pick a band name and run the program once. Here's an example test plan for this simple program (recall that Writing the Test Plan is the third of our problem-solving steps):

Test Case 1 : Checking Input and Output

Step	Input	Expected Results	Actual Results
1	The Clash for band name	Band Name : The Clash	

We should mention a couple of things here. First of all, each test case in a test plan represents one complete execution of the program; no more, and no less. The above test plan only has one test case because we only have to run the program once to fully test it. The Expected Results tell us what the program should do at each step if the program works as it's supposed to.

But why is the Actual Results column blank? Because we haven't written ANY code yet! We develop our test plan based on the structure of

the program, which we get from the algorithms we write and the resulting CFGs. Once we've written the code, we can complete the last of our problem-solving steps (Test the Code), which involves running the program and filling in the Actual Results.

4.2. Testing Selection Control Structures

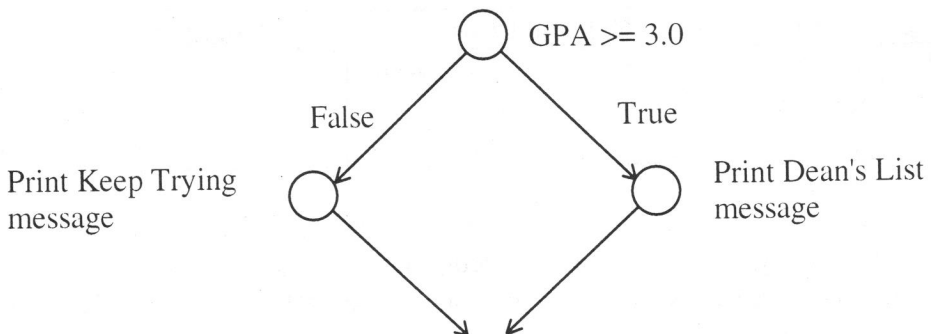
We've already seen that it's fairly easy to test a program with a sequential control structure. But suppose our program contains a selection control structure? Recall our second selection control structure example from the previous chapter:

Example 4.2 Printing a Dean's List or Keep Trying Message

Problem Description: Write an algorithm that will print a Dean's List message if the student's GPA is 3.0 or above, and will print a Keep Trying message if it's not.

Algorithm and CFG:

```
-- If GPA greater than or equal to 3.0
-- Print Dean's List message
-- Otherwise
-- Print Keep Trying message
```



Remember, when we use a selection control structure, we end up with two or more *branches*. Testing becomes more difficult if our CFGs contain branches. To thoroughly test such programs, we need to do two things:

1. Test every branch at least once,
2. Test boundary values in the boolean expression for the selection control structure, and

In our example, if the GPA is greater than or equal to 3.0, the program prints out a Dean's List message, otherwise the program prints the Keep Trying message. So how do we test every branch at least once? We run the program with a GPA ≥ 3.0 and make sure it prints the Dean's List message, then we run the program again with a GPA < 3.0 to make sure it prints the Keep Trying message. But what about testing boundary values in the boolean expression? Because the critical factor in the boolean expression is whether or not the GPA is < 3.0 or ≥ 3.0 , we should try values of 2.9, 3.0, and 3.1 for the GPA to make sure the program works properly for all of them¹. Of course, using these three values will end up testing both branches as well, so running the program three times (once with each of the values) should provide sufficient testing for this program.

Here's an example test plan for our program (again without actual results, because we haven't written the code yet):

Test Case 1 : Checking Left Branch, Boundary Value 2.9

Step	Input	Expected Results	Actual Results
1	2.9 for GPA	Keep Trying message	

Test Case 2 : Checking Right Branch, Boundary Value 3.0

Step	Input	Expected Results	Actual Results
1	3.0 for GPA	Dean's List message	

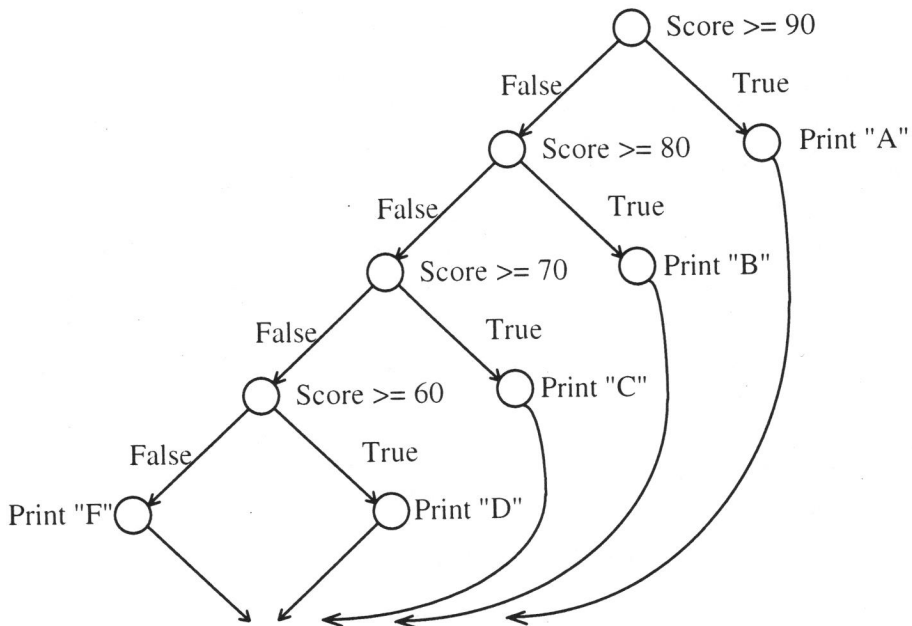
¹Of course, there are plenty of floating point numbers between 2.9 and 3.0, so it can be argued that we're not EXACTLY at the boundary with these values. For the programs in this book, however, testing within 0.1 of floating point boundaries will be sufficient.

Test Case 3 : Checking Boundary Value 3.1

Step	Input	Expected Results	Actual Results
1	3.1 for GPA	Dean's List message	

Because we need to run the program three times, we developed three separate test cases. Remember, we have to have one test case for each execution (or run) of the program. Of course, these one-step test cases are pretty simple, but as your programs get more complicated, so will your test plans.

Now, suppose we have an even more complicated program -- one that determines and prints a letter grade given a test score (assuming there's no curve!). The CFG would probably look something like this:



We now have lots of branches to test, and to test them all we need to run the program with a score for an A, a score for a B, a score for a C, a score for a D, and a score for an F. But it's not quite that easy, since we also have to test the boundary conditions for each boolean expression in the selection statements. We therefore should test the following scores during

testing: 91, 90, 89, 81, 80, 79, 71, 70, 69, 61, 60, and 59. We have to run the program 12 times to make sure it's working properly! Clearly, adding a little extra complexity to our program can make testing the program much harder.

4.3. Testing Iteration Control Structures

Recall that the iteration control structure (or loop) lets us execute certain steps in our algorithm multiple times. When a program contains loops, "completely" testing the program becomes impossible. Think about a single loop -- to really test it, you'd have to execute the loop body (the part after the "test" node) zero times, execute the loop body once, execute the loop body twice, ... you see where this is going, right? When the program contains loops, we need to compromise. The best way to test such programs is to execute each loop body zero times, one time, and multiple times. And of course we still need to test the boundary values for each boolean expression (and, occasionally, check combinations in the boolean expressions as well).

Let's revisit our iteration control structure example from the previous chapter:

Example 4.3 Getting a Valid GPA

Problem Description: Write an algorithm that will ask for a GPA until a valid GPA is entered.

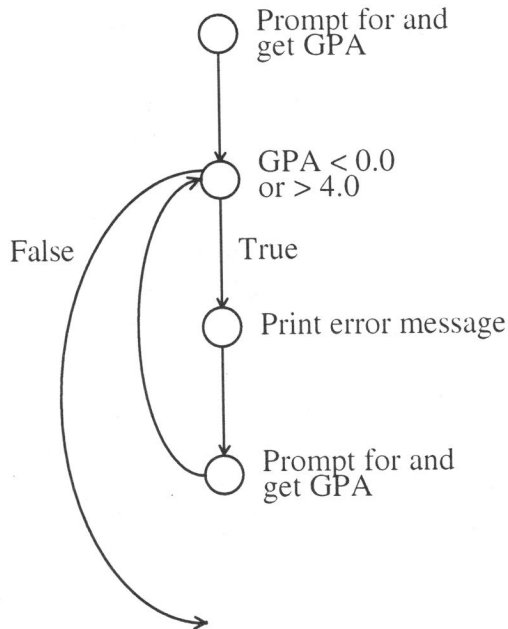
Algorithm and CFG:

```
-- Prompt for and get GPA
-- While GPA is less than 0.0 or greater than 4.0
  -- Print error message
  -- Prompt for and get GPA
```

The associated CFG can be found on the following page.

How do we execute the loop body (the nodes that print an error message and get a new GPA) zero times? By entering a valid GPA the first time we're asked. To execute the loop body one time, we enter an invalid

GPA followed by a valid one. To execute the loop body multiple times, we enter a few invalid GPAs followed by a valid one. To test the boundary values, we want to input the following values for GPA: -0.1, 0.0, 0.1, 3.9, 4.0, and 4.1. An example test plan is provided below.



Test Case 1 : Executing Loop Body 0 Times, Checking Boundary Value 0.0

Step	Input	Expected Results	Actual Results
1	0.0 for GPA	Execute and stop	

Test Case 2 : Executing Loop Body 1 Time, Checking Boundary Values -0.1 and 0.1

Step	Input	Expected Results	Actual Results
1	-0.1 for GPA	Error message, reprompt	
2	0.1 for GPA	Execute and stop	

Test Case 3 : Executing Loop Body Multiple Times, Checking Boundary Values 4.1 and 4.0

Step	Input	Expected Results	Actual Results
1	-0.1 for GPA	Error message, reprompt	
2	4.1 for GPA	Error message, reprompt	
3	4.0 for GPA	Execute and stop	

Test Case 4 : Checking Boundary Value 3.9

Step	Input	Expected Results	Actual Results
1	3.9 for GPA	Execute and stop	

Finally we have test cases with multiple steps!

4.4. Testing Combinations of Control Structures

Now suppose we have a program that has to both select and iterate? How do we test it? By combining what we've learned about testing selection and iteration statements. Consider our combination example from the last chapter:

Example 4.4 Getting a Valid GPA and Printing a Message

Problem Description: Write an algorithm that will ask for a GPA until a valid GPA is entered, then print a Dean's List message if the GPA is greater

than or equal to 3.0 or print a Keep Trying message if the GPA is less than 3.0.

Algorithm and CFG:

```
-- Prompt for and get GPA
-- While GPA is less than 0.0 or greater than 4.0
  -- Print error message
  -- Prompt for and get GPA
-- If GPA greater than or equal to 3.0
  -- Print Dean's List message
-- Otherwise
  -- Print Keep Trying message
```

The associated CFG can be found on the following page.

To test this program, we need to execute the loop body zero, one, and multiple times, execute each branch of the selection at least once, and make sure all boundary values are covered. An example test plan is given below.

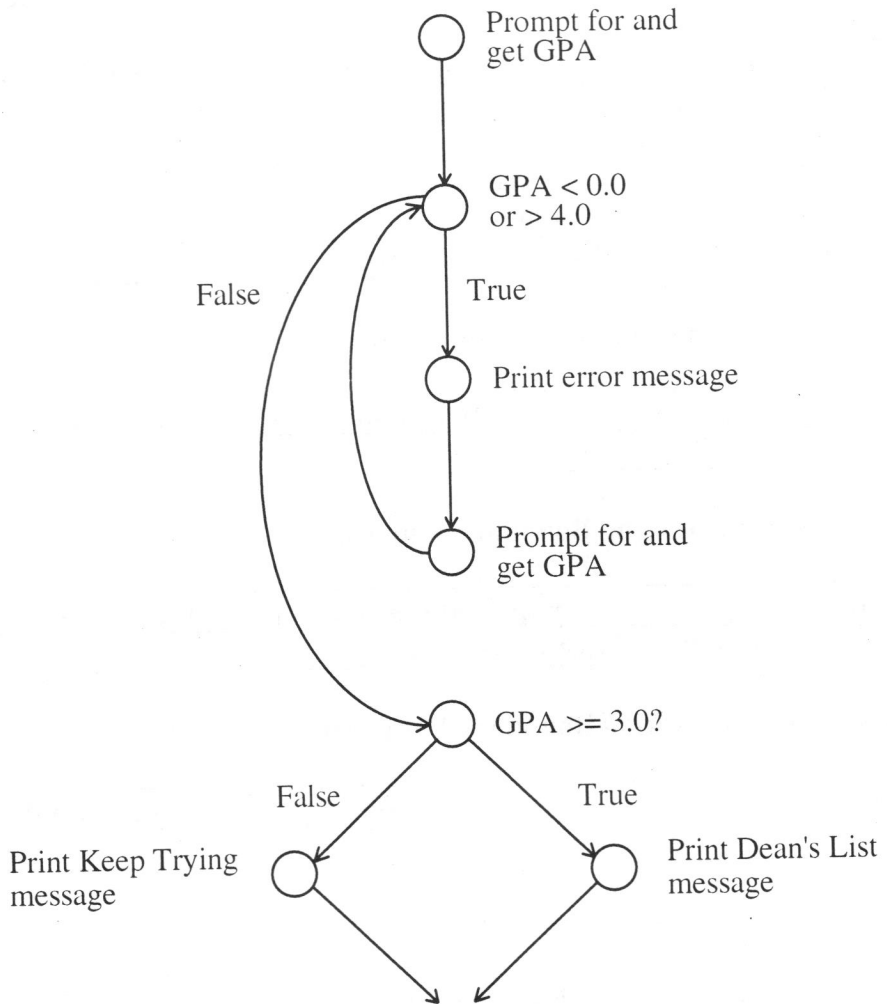
Test Case 1 : Executing Loop Body 0 Times, Left Branch of Selection, Checking Boundary Value 0.0

Step	Input	Expected Results	Actual Results
1	0.0 for GPA	Keep Trying message	

Test Case 2 : Executing Loop Body 1 Time, Checking Boundary Values -0.1 and 0.1

Step	Input	Expected Results	Actual Results
1	-0.1 for GPA	Error message, reprompt	
2	0.1 for GPA	Keep Trying message	

(The test plan is continued on the page following the CFG).



Test Case 3 : Executing Loop Body Multiple Times, Right Branch of Selection, Checking Boundary Values 4.1 and 4.0

Step	Input	Expected Results	Actual Results
1	-0.1 for GPA	Error message, reprompt	
2	4.1 for GPA	Error message, reprompt	
3	4.0 for GPA	Dean's List message	

Test Case 4 : Checking Boundary Value 3.9

Step	Input	Expected Results	Actual Results
1	3.9 for GPA	Dean's List message	

Test Case 5 : Checking Boundary Value 2.9

Step	Input	Expected Results	Actual Results
1	2.9 for GPA	Keep Trying message	

Test Case 6 : Checking Boundary Value 3.0

Step	Input	Expected Results	Actual Results
1	3.0 for GPA	Dean's List message	

Test Case 7 : Checking Boundary Value 3.1

Step	Input	Expected Results	Actual Results
1	3.1 for GPA	Dean's List message	

As you can see, the more complicated the program becomes, the harder it is to test it.

4.5. Completing the Test Plan

So far, our discussion has focused on using the program structure to figure out which inputs we should use in our test plan and what the expected results are for those inputs. This corresponds to the third step in our problem-solving process. After we write the code, we need to complete the final problem-solving step: Test the Code, which involves executing the program (using the test plan) to see if it behaves the way it's supposed to (and filling in the actual results portion of the test plan).

If all the actual results match the expected results (and you've done a thorough job developing your test cases), your program *probably* works the way it should. But what if your actual results don't match your expected results? There are two common scenarios:

1. Your expected results are correct, but you've written your code incorrectly.
2. Your code is correct, but you've determined your expected results incorrectly.

For most of us, the first scenario is much more common. When this is the case, you need to go back to your code, debug it (find and fix the errors in your code), then re-run the ENTIRE TEST PLAN. Why not just re-run the test case that failed? Because the changes you made to your code could change how the program performs on the other test cases, so you need to re-run all the test cases.

For the second scenario, you can simply fix the expected results in your test plan, but you should be absolutely sure that this is really the problem. Incorrect code is much more common than incorrect expected results.

Once you've accomplished any required debugging and your actual results match your expected results, your testing is complete.

4.6. Conclusions

To make sure a computer program does what it's supposed to, we need to perform thorough testing of that program. We can decide how to test the program without actually writing it -- all we need to know is the structure

of the program, which we can glean from the algorithms and resulting CFGs. Once we know the conditions under which the program selects or iterates, we can choose our input data and determine our expected results. After the code is completed, we can use the test plan to make sure the actual results match the expected results (e.g., that the program works properly). You should always thoroughly test your programs before you turn them in for a grade, because you can be sure your teacher will before assigning your grade!

4.7. Exercises

1. Draw a CFG for the following algorithm:

```
-- Prompt for and get first number
-- Prompt for and get second number
-- If first number is greater than second number
    -- Print first number is higher
-- Otherwise
    -- Print second number is higher
```

2. Write a test plan to test the algorithm in Exercise 1.

3. Draw a CFG for the following algorithm:

```
-- Prompt for and get test percentage
-- While test percentage is less than 0 or greater than
  100
    -- Print error message
    -- Prompt for and get test percentage
```

4. Write a test plan to test the algorithm in Exercise 3.

5. Draw a CFG for the following algorithm:

```
-- Initialize continue to Yes
-- While continue is Yes
  -- Prompt for and get an age
  -- If the number is less than 21
    -- Print "Still a minor" message
  -- Otherwise
    -- Print "Adult" message
  -- Prompt for and get continue
```

6. Write a test plan to test the algorithm in Exercise 5.

Chapter 5. Your First Ada Program

Well, for a book with Ada in the title we sure haven't talked about Ada much! That's because it's very important that you understand the *process* we go through to solve problems before you actually try to implement those problem solutions. Now that we've laid those foundations in the previous chapters, it's time to look at our first Ada program. As we learn about Ada, we'll learn how we can implement our problem solutions using Ada.

5.1. What Does an Ada Program Look Like?

The syntax for a "typical" Ada program is provided below. We'll discuss all the parts in further detail below, so don't worry if you don't understand them all right away.

Ada Program Syntax

```
<main program comment block>

<withs and uses of packages>

procedure <main program name> is

    <type declarations>

    <constant declarations>

    <procedures in the program>

    -----
    -- START OF MAIN PROGRAM
    -----

    <variable declarations>

begin

    <executable statements>

end <main program name>;
```

Since we'll be providing lots of "Syntax Boxes" (like the one above) in the coming chapters, we should discuss the notation a bit. When we list items between `<` and `>` (like `<main program comment block>`, for example), that means you, the programmer, decide what goes there. When we list words without any special notation (like `procedure`, `is`, `begin`, and `end`), those words need to appear EXACTLY as written. That's because these are special words in Ada (they're called *reserved words*), so you need to use them just as they appear in the syntax boxes.

OK, let's discuss each of the parts of an Ada program in more detail.

5.2. Comments

The first item in the syntax box above is the main program comment block. A comment is a double dash, `--`, followed by whatever descriptive text you choose to add after the double dash. Although comments aren't actually required by the language, it's always a good idea to provide documentation within your programs. The main program comment block should contain the author's name, a description of the program, and an algorithm for the program. Here's an example:

```
-----
--
-- Author : Axl Rose
-- Description : This program simply prints "Hello, world,
--               It's November - is it raining?"
-- Algorithm :
--               Print message to the screen
--
-----
```

There are also two other kinds of comments we include in the programs we write. For each procedure in our solution (more about procedures later), we write a comment block similar to the one for the main program. Procedure comment blocks contain the name of the procedure, a description of the procedure, and an algorithm for the procedure. We'll see lots of examples of these in a few chapters.

The last kind of comment we include in our program is called the line comment. Every few lines of Ada code you write should have a comment above them explaining what those lines are supposed to do. Your algorithm should describe exactly what the program is doing at each step, so it's not necessary to provide a line comment for every line of code. There are certainly different philosophies about how many line comments you should include, so pick a style that seems effective to you (and your teacher).

5.3. Using Other Packages

The next part of an Ada program is the place where we with and use packages. Packages are collections of useful Ada code that someone else has already written (so that you don't have to!). For example, you'll probably always want to have access to procedures that let you print characters and strings of characters to the screen. These are found in the `Ada.Text_IO` package; here's what you should include in your program:

```
with Ada.Text_IO;
use  Ada.Text_IO;
```

The with clause tells the compiler that you want access to the procedures in the `Ada.Text_IO` package. The use clause is really for your convenience, since it means you won't be required to add `Ada.Text_IO.` as a prefix to every call on a procedure in the package. We'll introduce you to other packages provided in Ada as we need them.

5.4. Identifiers

Whenever we need to name something in our program (like for `<main program name>`), we need to use an identifier (which is just another word for name). Ada has a few rules for which identifiers are legal and which aren't.

Identifiers have to start with a letter, which can be followed by any number of underscores, letters, and numbers. Identifiers can't contain two consecutive underscores, though, and they can't be reserved words

(remember, those special Ada words like *procedure*, etc.). Here are some examples:

Legal Identifiers: GPA, Last_Name, Band_42, A_Really_Long_Name
 Illegal Identifiers: 2_Names, My_%%\$_Chores, Big_Space

5.5. Variables and Constants

Computers were originally designed to do mathematical calculations, so it shouldn't come as a surprise that many of the programs you write will also do calculations. You should remember from your math and science classes that math equations contain a number of different things. For example, consider the equation for the area of a circle:

$$\text{Area} = \pi r^2$$

Area and r are the variables in the equation, while π is a constant in the equation. Ada lets us declare the required variables and constants as follows:

```
Pi      : constant Float := 3.1416;
Area    : Float;
Radius  : Float;
```

Don't worry about what `Float` means yet -- we'll get to data types in the next chapter. At this point, you should simply realize that Ada lets us declare both variables and constants.

5.6. Simple Input and Output

It's nice to have a program that goes off and crunches numbers, but it's even nicer to have the program tell us the answer after it figures it out! And how does the program get the numbers in the first place? That's what input and output are all about. In this section, we'll talk about how we do input and output for characters and strings of characters (remember the

Ada.Text_IO package?). Input and output for numbers can be a little different, so we defer that discussion to the next chapter.

To put some output to the screen, we use the `Put` procedure (nice name, huh?). For example to output the user's first initial to the screen, we'd use:

```
Put (Item => "First initial : ");
Put (Item => First_Initial);
New_Line;
```

The first `Put` prints the words "First initial : " (without the quotes) to the screen. We always want our programs to say what they're printing, and that's what the first `Put` does. The second `Put` prints out the value of `First_Initial`, and the `New_Line` moves the cursor to the next line on the screen.

To get some input from the user, we use the `Get` procedure (someone really thought about these names!). For example, to read in the user's first initial from the keyboard into a variable called `First_Initial`, we'd use:

```
Put (Item => "Please enter first initial : ");
Get (Item => First_Initial);
Skip_Line;
```

The `Put` prints something called a *prompt*, since it prompts the user for input. The `Get` then gets the user's input, with the `Item` in the `Get` simply telling the compiler that `First_Initial` is the thing (or item) we want to get. The `Skip_Line` simply throws away anything else the user types up to the time they press the <enter> key. Although in some cases the `Skip_Line` isn't actually necessary, it's usually a good idea to include one after EVERY `Get` you do.

Many people seem to have a hard time deciding between `Skip_Line` and `New_Line` when they write their programs. If you simply remember that `Skip_Line` is for input and `New_Line` is for output, you should be fine.

5.7. A Matter of Style

While beginning programmers tend to think that a program that works properly is perfect, more experienced programmers also recognize the importance of style. Good programming style tells us to use variable names that are descriptive (like `First_Initial` instead of `FI`) and to use capital letters at the start of variable names and underscores between words in those names. It also tells us to use proper indentation, good commenting, and to include "white space" (blank lines) in our programs. Following these style guidelines makes your programs easier to read and understand.

Like most style matters, programming style can be largely a matter of taste. We've selected a particular style for all the examples in this book, but your teacher may want you to use a different style. In any case, using reasonable, consistent style guidelines will help you develop better code.

5.8. Putting It All Together

Let's go through the entire problem-solving process for a simple problem. Here's the problem description:

Print the lines:

Hello, world,
It's November - is it raining?

to the screen.

Understand the Problem

There's not much to worry about here, since the problem seems to be easy to understand.

Design a Solution

This problem is small enough that we don't need to worry about breaking it down into subproblems. We can therefore go right into our algorithm, which really only includes one step:

```
--      Print message to the screen
```

So we're ready to move on to our next step.

Write the Test Plan

Since this program won't have any user input, all we have to do is run it to make sure it prints out the required message.

Test Case 1 : Checking Message

Step	Input	Expected Results	Actual Results
1	None	Rain message	

Write the Code

Here's the completed code for the program:

```
-----
--
-- Author : Axl Rose
-- Description : This program simply prints "Hello, world,
--      It's November - is it raining?"
-- Algorithm :
--      Print message to the screen
--
-----

with Ada.Text_IO;
use  Ada.Text_IO;

procedure Rain_Message is

-----
--  START OF MAIN PROGRAM
-----

begin
```

```
--      Print message to the screen
Put (Item => "Hello, world,");
New_Line;
Put (Item => "It's November - is it raining?");
New_Line;
New_Line;
```

```
end Rain_Message;
```

Test the Code

Now we simply run our program to make sure we get the results we expected. We fill in the actual results portion of our test plan, yielding the following:

Test Case 1 : Checking Message

Step	Input	Expected Results	Actual Results
1	None	Rain message	Rain message

And that's it -- we've completed our first Ada program!

Chapter 6. Data Types, Variables, and Constants

In the last chapter, we made some cryptic references to data types (and we told you not to worry about them until later). Well, it's later! Ada is called a *strongly-typed language*, which means we always have to decide what data type each variable and constant we declare will be, and we're generally not allowed to "mix" different data types together in mathematical operations². Data types are very important in Ada, so let's take a closer look.

6.1. What's a Data Type?

Whenever we need to store something (like a GPA, or a name, or the value of Pi, etc.) in our program, we need to declare a variable or constant. So what are we really doing when we do this declaration? We're setting aside a "box in memory" to hold that variable or constant. For a variable, the box starts out containing whatever "garbage" was left there by the last program using that box in memory, and we can change what's in that box (called the *value* of the variable) as many times as we want as the program executes. For a constant, the box contains whatever we said the constant's value would be, and we're never allowed to change it as the program executes. When we decide to declare a variable or constant in our program, we use the syntax on the following page.

The variable or constant names need to be legal identifiers (remember we discussed the rules for those in the previous chapter). The data type for a variable or constant tells us two things:

1. What values the variable/constant can have, and
2. What operations are valid for the variable/constant.

Let's look at these a little more closely in the context of one of our variables from the previous chapter.

²There are actually ways in Ada to do some "mixing", but we won't go into them until later in the book.

Variable and Constant Declaration*Variables*

<variable name> : <data type>;

<variable name> - the name of the variable.

<data type> - the data type for that variable.

Constants

<constant name> : constant <data type> := <value>;

<constant name> - the name of the constant.

<data type> - the data type for that constant.

<value> - the value of the constant.

```
Area : Float;
```

The Area variable is declared as a Float (or floating point number, which is a number with a decimal point, like 3.2, 12.75, etc.), which tells us the only values this variable can have are floating point numbers. In other words, we can't store a character, a string of characters, or any other value that's not a floating point number in this variable.

We also know which operations are valid for this variable. We know we can add this variable to another floating point number, but we can't add it to a character (remember our rule about not mixing data types in mathematical operations). We can also include the variable in any other mathematical operations that are valid for floating point numbers.

Similarly, if we wanted a constant for Pi, we could use:

```
Pi : constant Float := 3.1416;
```

and we could use Pi in floating point operations in our program.

6.2. Common Data Types

It turns out that there are some data types that are commonly used. We discuss each of them below.

Integer

Integers are whole numbers: -2, 0, and 42 are some examples. We might use an integer variable to count things or to store the number of points a student gets on a test.

The valid operations for integers are generally as you'd expect: addition, subtraction, and multiplication work in the usual ways. Division, however, is a little different. When we divide one integer by another in Ada, the result is always an integer as well. For example, 6 / 4 will give us 1, not the 1.5 you might expect. Just think of this as the math you did a long time ago, where you'd do integer divisions to get the result of the division and a remainder. Using that kind of math, 6 divided by 4 is 1 remainder 2. Ada also lets us calculate the remainder: 6 rem 4 will give us 2.

Float

Floats are also numbers, but they have a decimal part as well as a whole number part: -3.7, 0.00001, 5.0, and 483.256 are some examples. We might use a float variable to add up a set of floats or to hold a student's class percentage.

The valid operations for floats are as you'd expect, with addition, subtraction, multiplication, and division defined in the normal way.

Character

A variable or constant declared as a character can hold a single character (like A, 7, #, etc.). In many cases, this character will be a letter or a digit, but it can also be a space, a punctuation mark, or other characters. We might use a character variable to store a menu choice or a student's first initial.

In this book, we won't worry too much about the valid operations for characters. As long as we don't try to do anything strange (like multiplying two characters!) we should be fine.

String

A string is simply a set (or string) of characters: ftatrwsy, Clash, and triathlon are some examples. We might use a string variable to store a student's last name.

It turns out that it's not enough to tell the Ada compiler that we want a variable of type String; we also have to tell it EXACTLY how many characters are in the string. For example:

```
Last_Name : String (1..15);
```

declares the variable Last_Name as a string of exactly 15 characters. For some very good, but somewhat complicated, reasons, if we declare another variable:

```
First_Name : String (1..15);
```

the two variables are NOT the same type in Ada! We won't worry about why that's true, but we will show you how to make them have the same type.

We start by first defining a new type as follows:

```
subtype String_15 is String (1..15);
```

The subtype just tells the compiler that a String_15 is a special kind of string (one for which we already know how many characters it has). We can then declare our variables as:

```
First_Name : String_15;  
Last_Name  : String_15;
```

and they'll both be the same type. Although it's not always necessary to do things this way, if you have multiple string variables that have the same number of characters, it's better to define the subtype then declare the variables of that subtype.

Since strings are just composed of characters, we won't worry about the valid operations for them (but we also won't try to do anything strange with them). Ada actually does provide ways to combine strings and characters, but those operations are beyond the scope of this book.

true or false

Boolean

A Boolean variable (or constant) can only have one of two values: true or false. While this might not seem particularly useful to you right now, you'll find that Boolean variables are handy as flags to tell us whether or not to do certain things.

The valid operators for Boolean variables are probably somewhat unfamiliar to you: the ones we'll use in this book are called *and*, *or*, and *not*. Let's consider each of these.

When we *and* two Boolean operands, we're really trying to find out if they're both true. That means that:

False and False = False

False and True = False

True and False = False

True and True = True

In other words, the result of an *and* will only be true if both operands are true.

When we *or* two Boolean operands, we're really trying to find out if one (or both) of them is true. That means that :

False or False = False

False or True = True

True or False = True

True or True = True

In other words, the result of an *or* will be true if one or both of the operands is true.

The last Boolean operator we'll consider, *not*, is slightly different because it's a unary operator (it only takes one operand). The idea behind *not* is that it will "flip" the operand; that means that:

not False = True

not True = False

That's all you need to know about valid operators on Booleans.

Operations

Recall that a data type tells us the valid values and operations for variables and constants of that data type. For easy reference, here's a table of the operations you're most likely to use in an introductory course:

<u>Operator</u>	<u>Operation</u>	<u>Data Types</u>
+	Add	Integer, Float
-	Subtract	Integer, Float
*	Multiply	Integer, Float
/	Divide	Integer, Float
rem	Remainder	Integer
=, /=	Equal, Not Equal	Integer, Float, Character, String, Boolean
<, >, <=, >=	Less Than, Greater Than, Less Than or Equal To, Greater Than or Equal To	Integer, Float, Character, String, Boolean
and, or, not	And, Or, Not	Boolean

6.3. Choosing Between Variables and Constants

Let's think about variables and constants for a moment. Variables are things that can change, or vary; therefore, any variables we declare in our program can be changed as many times as we want as the program executes. Constants, on the other hand, are unchanging, or constant; therefore, once we declare a constant in our program, we can't change its value in other parts of the program. If we need a box in memory to store some value or multiple values during the execution of the program, we should declare that box as a variable (because its contents change). If we need a box in memory to hold some value that will never change during program execution, we should declare that box as a constant (because its contents don't change).

As we showed in the Ada Program Syntax box in the last chapter, the main program variables are declared just above the begin for the main

program, while constants are declared near the top of the program. We put the main program variables down near the begin to make sure none of our procedures (more about procedures later) change our variables "by mistake". Since constants aren't allowed to change, we put them at the top so that they're available for use by the entire program.

6.4. Giving Variables a Value

We already know how to give constants a value -- we do that when we declare the constant. But how do we give variables a value? There are actually two ways we can do this: with an assignment statement, and with a Get. Let's do an example that uses both these techniques.

Example 6.1 Reading a Circle's Radius and Calculating Its Area

Problem Description: Write a code fragment that will read in a circle's radius and calculate its area.

Algorithm: Here are the two steps we need to do:

```
-- Prompt for and get circle's radius
-- Calculate area as Pi * Radius ** 2
```

And when we implement the algorithm in Ada, we get:

```
-- Prompt for and get circle's radius
Put (Item => "Please enter the circle's radius : ");
Get (Item => Radius);
Skip_Line;

-- Calculate area as Pi * Radius ** 2
Area := Pi * Radius ** 2;
```

When we Get the radius, the computer takes whatever number was typed on the keyboard and stores that value into the Radius variable -- that's one way we give a variable a value.

Assignment statements are a bit more complicated (though not much), so let's look at the syntax first.

Assignment Statements`<variable name> := <expression>;``<variable name>` - the name of the variable.`<expression>` - some mathematical equation.

So what really happens with an assignment statement? First, the computer evaluates the expression to the right of the `:=`. In other words, it figures out the value of the right-hand side of the assignment statement. It then takes this value and puts it in the variable on the left of the `:=`. Here are some examples using a variety of data types:

<u>Data Type</u>	<u>Assignment Statement</u>
Integer	Age := 36;
Float	GPA := 3.99;
Character	First_Initial := 'A';
String (10 characters)	Last_Name := "Chamillard";
Boolean	Like_Spinach := True;

One warning about assignment statements -- because Ada is strongly typed, the data type of the expression MUST match the data type of the variable. In other words, if the expression evaluates to a Float value, ~~you'd better~~ be putting that value into a Float variable!

In our example, we're calculating the area of the circle using the standard equation (and because Pi and Radius are Float, the result is a Float) and putting the answer into Area (a Float variable). The ** in the equation simply means "raise to the power of". Notice that we have the integer 2 in the exponent; exponents need to be integers in Ada, so we're allowed to include this integer in our equation even though the other values are floating point numbers.

So there you have it -- the two ways to get a value into a variable.

6.5. Input/Output of Different Data Types

In the last chapter, we talked about simple input and output of characters and strings. In this section, we'll cover input and output of all the data types discussed above.

Integer I/O

Before we can do any input or output of integers, we need to with and use the package that contains the Ada code that lets you read integers from the keyboard and print them to the screen. These are found in the `Ada.Integer_Text_IO` package; here's what you should include in your program:

```
with Ada.Integer_Text_IO;
use  Ada.Integer_Text_IO;
```

Now that we have access to the appropriate procedures, we can start doing input and output of integers.

To get integer input from the user, we use the `Get` procedure. This is a different procedure from the `Get` we discussed in the last chapter (for characters and strings) because this one gets integers instead, but it works exactly the same way. For example, to read in the user's points on a test from the keyboard into an Integer variable called `Test_Points`, we'd use:

```
Put (Item => "Please enter test points : ");
Get (Item => Test_Points);
Skip_Line;
```

To output an integer value to the screen, we use the `Put` procedure. Here's an example that outputs the value of `Test_Points` to the screen:

```
Put (Item => "Test Points : ");
Put (Item => Test_Points,
    Width => 3);
New_Line;
```

The `Item` tells the computer to print out the value of `Test_Points`, and the `width` tells the computer how many spaces that output should take. To see how `width` works, let's look at some examples (with `width` set to 3):

<u>Value</u>	<u>Output</u>
100	100
89	89
3	3
1025	1025

When the `width` exactly matches the value being printed (like for 100), the output is exactly as you'd expect. When the `width` is larger than the value being printed (like for 89 and 3), the computer prints leading spaces, then the value. When the `width` is smaller than the value being printed (like for 1025), the computer ignores `width` and prints the entire value. It does this so we never "lose" any of our number in the output (what would you print in 3 spaces - 102? 025?). You'll find that `width` really helps us create nice-looking output, whether we're simply printing a single integer or a table of integers.

Float I/O

As we did for integers, we need to `with` and `use` the package that contains the procedures that let you read floating point numbers from the keyboard and print them to the screen. These are found in the `Ada.Float_Text_IO` package; here's what you should include in your program:

```
with Ada.Float_Text_IO;
use  Ada.Float_Text_IO;
```

To get float input from the user, we use the `Get` procedure. Again, this `Get` works the same way as the `Get` procedures we've discussed so far. For example, to read in the user's class percentage from the keyboard into a `Float` variable called `Class_Percentage`, we'd use:

```
Put (Item => "Please enter class percentage : ");
Get (Item => Class_Percentage);
Skip_Line;
```

To output a floating point value to the screen, we use the `Put` procedure. Here's an example that outputs the value of `Class_Percentage` to the screen:

```
Put (Item => "Class percentage : ");
Put (Item => Class_Percentage,
     Fore => 3,
     Aft  => 2,
     Exp  => 0);
New_Line;
```

The `Item` tells the computer to print out the value of `Class_Percentage`, the `Fore` tells the computer how many spaces to output to the left of the decimal point, the `Aft` tells how many spaces to output to the right of the decimal point, and the `Exp` tells how many spaces to use for the exponent in scientific notation. Using 0 for `Exp` gives output that's not in scientific notation. To see how these work, let's look at some examples (with `Item` set to 89.3572):

<u>Fore</u>	<u>Aft</u>	<u>Exp</u>	<u>Output</u>
1	2	0	89.36
2	1	0	89.4
3	5	0	89.35720
2	2	3	8.94E+01
2	2	1	8.94E+1

Notice that `Fore` works very much like width for integers, including the fact that `Fore` never lets us "lose" any of the whole number part of our output. When `Aft` is smaller than the decimal portion of the number, the computer rounds to fit the number in the required number of spaces. When `Aft` is larger than the decimal portion of the number, the computer adds zeros to the end. When `Exp` is 0, the number is printed "normally" (not in scientific notation). When `Exp` is greater than 0, it tells how many spaces to use for the exponent, but in this case it will always use enough spaces to print the `E`, the sign for the exponent, and the entire exponent.

Character and String I/O

We started talking about input and output of characters and strings in the last chapter. Recall that the procedures required to do this are in the `Ada.Text_IO` package, so we need to include:

```
with Ada.Text_IO;
use  Ada.Text_IO;
```

in any programs that need to do this (every program we write in this book, anyway).

As we've already discussed, we can use the `Get` procedure to do input of characters and strings, so here's one way to read in the user's first initial:

```
Put (Item => "Please enter first initial : ");
Get (Item => First_Initial);
Skip_Line;
```

and similarly for last name:

```
Put (Item => "Please enter last name : ");
Get (Item => Last_Name);
Skip_Line;
```

The `Get` for the character variable `First_Initial` works exactly as you'd expect, but the `Get` for the string variable `Last_Name` requires a bit of explanation. Recall that `Last_Name` is a string of EXACTLY 15 characters. What if the user's name is more than 15 characters long? The `Get` reads in the first 15 characters in the last name and the `Skip_Line` throws away the rest! What if the user's last name is less than 15 characters -- what happens when they enter their name (Smith, for example), and press the <enter> key? The computer appears to "hang", waiting for the user to enter the other 10 characters! The user can enter spaces if they want (or any other characters), but pressing the <enter> key (even 10 times) won't help. This is somewhat awkward for the user, so we can make this a little nicer by using the `Get_Line` procedure. We use it as follows (Assume we've also declared an integer variable called `Name_Length`):



```
Put (Item => "Please enter last name : ");
Get_Line (Item => Last_Name,
          Last => Name_Length);
```

When we use `Get_Line`, we still read into the string variable, but `Last` also tells us how many characters the user entered. This makes it so the user doesn't have to enter trailing spaces to fill the string, and also makes our output look nicer.

To output characters and strings, we can use the `Put` procedure we discussed in the last chapter, so here's one way to output the user's first initial:

```
Put (Item => "First initial : ");
Put (Item => First_Initial);
New_Line;
```

and similarly for last name:

```
Put (Item => "Last name : ");
Put (Item => Last_Name);
New_Line;
```

Here's some (slightly) more complicated output:

```
Put (Item => "Thanks, Mr. ");
Put (Item => Last_Name);
Put (Item => ". Have a nice day!");
New_Line;
```

When we use `Put` for a string this way, the computer outputs the entire string, including trailing spaces. That means we can end up with output that looks like:

```
Thanks, Mr. Smith           . Have a nice day!
```

We can make this look much nicer, though if we happen to know the length of the string (say from `Name_Length` from the `Get_Line` above). If we know the length of the string, we can slightly modify the output of the last name to:

```
Put (Item => Last_Name(1..Name_Length));
```

to get much nicer output:

★ Thanks, Mr. Smith. Have a nice day!

★ By putting (1..Name_Length) after the name of our string variable, we've told the computer to print all the characters between (and including) the 1st character and the Name_Lengthth character. That means it prints the string value in exactly the right number of spaces, giving us much nicer output.

There is one other procedure we can use to output characters and strings -- Put_Line. Put_Line works just like a Put followed by a New_Line, so we could replace the two lines:

1 ↓ Put (Item => ". Have a nice day!");
2 ↓ New_Line;

with

3 ↓ Put_Line (Item => ". Have a nice day!");

factor/cond

to get the same output.

Boolean I/O

Boolean I/O can be a little tricky, since we need to do something called *instantiating a generic*. We won't really worry about the low-level details, since generics are beyond the scope of this book, but we will show you the syntax for doing this. All we really have to do is say we want a new package to do Boolean I/O (we'll call the new package Boolean_IO), then we use the new package:

★ package Boolean_IO is new Ada.Text_IO Enumeration_IO (
Enum => Boolean);
use Boolean_IO;

And that's all there is to it! We can simply use `Get` and `Put` for input and output of Boolean values, and the compiler will use the new `Boolean_IO` package to perform those operations.

6.6. Putting It All Together

Now let's go through the entire problem-solving process for a problem that needs to use constants, variables, input and output. Here's the problem description:

Read in the user's first initial and last name. Read in the radius of a circle, using the user's name in the prompt for the radius. Calculate the area of the circle, then print out the user's first initial and last name, the radius, and the area of the circle.

Understand the Problem

Do we understand the problem? We may not understand **HOW** to do everything yet, but do we understand **WHAT** our program needs to do? What if the user enters a negative radius? At this point, we'll just accept it and calculate the area anyway (we'll learn how to check for this later). Let's move on to the next step.

Design a Solution

Remember, this consists of two steps. The first step is to use top-down decomposition to break the problem into smaller subproblems. The second step is to write detailed algorithms to solve each of the subproblems. We haven't learned about procedures yet (next chapter!), so we'll go straight to writing a detailed algorithm. Here's one such algorithm:

```
-- Print introduction
-- Prompt for and get first initial
-- Prompt for and get last name
-- Prompt for and get radius
-- Calculate area as Pi * radius ** 2
-- Print out first initial
-- Print out last name
-- Print out radius
-- Print out area
```

Now we're ready to move on to our next step.

Write the Test Plan

Because this is a sequential algorithm (it doesn't contain any selection or iteration statements), we really only need one test case:

Test Case 1 : Checking Input, Calculation, and Output

Step	Input	Expected Results	Actual Results
1	G for first initial	Prompt for last name	
2	Thoroughgood for last name	Prompt for radius	
3	1.0 for radius	Print G for first initial, Thoroughgood for last name, 1.00 for radius, and 3.14 for area	

Write the Code

Because we did such a careful job with our algorithm development, writing the code is easy. Here's what it looks like:

```
-----
--
-- Author : Yngwie Malmsteen
-- Description : This program reads in the user's first
--               initial and last name. It then reads in the radius of
--               a circle, using the user's name in the prompt for the
--               radius. It calculates the area of the circle, then
--               prints out the user's first initial and last name, the
--               radius, and the area of the circle.
-- Algorithm :
--   Print introduction
--   Prompt for and get first initial
--   Prompt for and get last name
--   Prompt for and get radius
--   Calculate area as  $\text{Pi} * \text{radius} ** 2$ 
--   Print out first initial
```


62 Chapter 6

```
--      Print out last name
--      Print out radius
--      Print out area
--
```

```
-----
with Ada.Text_IO;
use  Ada.Text_IO;
```

```
with Ada.Integer_Text_IO;
use  Ada.Integer_Text_IO;
```

```
with Ada.Float_Text_IO;
use  Ada.Float_Text_IO;
```

```
procedure Calculate_Area is
```

```
    Pi : constant Float := 3.1416;
```

```
-----
--  START OF MAIN PROGRAM
-----
```

```
First_Initial : Character;
Last_Name     : String(1..15);
Name_Length   : Integer;
Radius        : Float;
Area          : Float;
```

```
begin
```

```
--      Print introduction
Put (Item => "This program reads in the user's first ");
Put (Item => "initial and last name.");
New_Line;
Put (Item => "It then reads in the radius of a ");
Put (Item => "circle.  It calculates ");
New_Line;
Put (Item => "the area of the circle, then prints out ");
Put (Item => "the user's first initial ");
New_Line;
Put (Item => "and last name, the radius, and the area ");
Put (Item => "of the circle.");
```

```

New_Line;
New_Line;

--    Prompt for and get first initial
Put (Item => "Please enter your first initial : ");
Get (Item => First_Initial);
Skip_Line;

--    Prompt for and get last name
Put (Item => "Please enter your last name : ");
Get_Line (Item => Last_Name,
          Last => Name_Length);

--    Prompt for and get radius
Put (Item => "Please enter a radius, ");
Put (Item => Last_Name(1..Name_Length));
Put (Item => " : ");
Get (Item => Radius);
Skip_Line;

--    Calculate area as Pi * radius ** 2
Area := Pi * Radius ** 2;

--    Print out first initial
New_Line;
Put (Item => "First Initial : ");
Put (Item => First_Initial);
New_Line;

--    Print out last name
Put (Item => "Last Name      : ");
Put (Item => Last_Name(1..Name_Length));
New_Line;

--    Print out radius
Put (Item => "Radius          : ");
Put (Item => Radius,
     Fore => 1,
     Aft  => 2,
     Exp  => 0);
New_Line;

```

```

--      Print out area
Put (Item => "Area           : ");
Put (Item => Area,
     Fore => 1,
     Aft  => 2,
     Exp  => 0);
New_Line;

end Calculate_Area;

```

You should notice that we with and use all the appropriate packages, declare and use both constants and variables, and make sure both our prompts and our output look nice.

Test the Code

So now we need to run our test plan to make sure the program works. When we run the program above and fill in the actual results in our test plan, we get the following:

Test Case 1 : Checking Input, Calculation, and Output

Step	Input	Expected Results	Actual Results
1	G for first initial	Prompt for last name	Prompts for last name
2	Thoroughgood for last name	Prompt for radius	Prompts for radius
3	1.0 for radius	Print G for first initial, Thoroughgood for last name, 1.00 for radius, and 3.14 for area	Prints G for first initial, Thoroughgood for last name, 1.00 for radius, and 3.14 for area

6.7. Common Mistakes

Using := to Declare Variables

Some programmers confuse the : required between a variable name and its data type and the := used to assign a value to a variable or constant. Using a := where a : goes will result in a compilation error.

Assuming Variables Start With "Nothing" or 0 In Them

Variables do NOT have a pre-defined value before the programmer gives them one. You can NOT assume that variables start at 0 (or nothing, whatever that means) -- the initial value of a variable is whatever was left in that memory location before you ran your program.

Mixing Types (Especially Floats and Integers)

One very common mistake is for programmers to mix data types in expressions. For example, using:

```
Circumference := 2 * Pi * Radius;
```

will result in a compilation error (assuming Pi and Radius are defined as Float as we'd expect). The above assignment tries to multiply an integer by two floating point numbers, and because Ada is strongly typed, we're generally not allowed to mix these types in our math. The obvious solution is to use:

```
Circumference := 2.0 * Pi * Radius;
```

instead.

Not Completely Filling String On Input Using Get

When our program is reading into a string variable using Get (rather than Get_Line), that variable holds a specific number of characters (i.e., 15 for the last name in our example above). When the user enters a value for that string, they MUST enter at least 15 characters. If they enter more, the extra characters will be discarded, but if they enter less the program will simply wait for them to enter the remaining characters. Pressing the <enter> key doesn't help, since enters don't count as characters in the string. This can be very confusing for the user, since it appears that the program has become "stuck."

Trying to Use Put_Line or Get_Line for Floats and Integers

Put_Line and Get_Line are NOT defined for Floats and Integers (only for strings). To get the same behavior for Floats and Integers, you should use a Put followed by a New_Line or a Get followed by a Skip_Line;

6.8. Exercises

1. Declare variables or constants for the following:
 - a. Points on a test
 - b. Test percentage
 - c. Letter grade (no + or - grades)
 - d. Letter grade (with + and - grades)
 - e. Hometown
 - f. Home state
 - g. Whether or not the user likes triathlons
2. Write a program to read in 3 test percentages and calculate the average percentage.
3. Write a program to read in the user's first name, middle initial, and last name, then print a message using these values. The output should NOT have any extra spaces before or after each value.

Chapter 7. Procedures

Remember that designing your solution to a problem consists of two steps -- breaking the problem into smaller subproblems, then writing a detailed algorithm for each of those subproblems. Because this process is so essential to problem-solving, Ada gives us a construct we can use to implement each of our subproblem solutions. That construct is called a procedure.

Procedures are useful for a number of reasons. They let us concentrate on one small part (the subproblem) of a problem at a time, making it easier to solve that small part. If we have to do something multiple times, we simply write the procedure once and call it multiple times (more about calling procedures later in this chapter). And in many cases, we can use a procedure we've already written to write a new procedure that solves a similar problem.

These procedure things sound pretty useful; let's figure them out. Before we jump into them, though, here's a handy syntax box showing how the entire procedure looks:

Procedure Syntax

```
<procedure header>
```

```
    <local variables and constants>
```

```
begin
```

```
    <procedure body>
```

```
end <procedure name>;
```

<procedure header> - contains the procedure name and parameters.

<local variables> - variables used in the procedure (but nowhere else).

<procedure body> - the code the procedure executes.

<procedure name> - the name of the procedure.

7.1. Deciding Which Procedures to Use

Great, you're convinced that procedures are useful and you want to learn more (or your teacher assigned this chapter as a reading, and you're trying to pass the course). The first question we need to ask is "Which procedures should I use to solve my problem?" But this isn't really a procedures question, is it? It's really a question about problem-solving -- how do I break my problem down into reasonable subproblems?

We discussed problem solving way back in Chapter 2, and we described top-down decomposition in Chapter 3. Now that you know a little more about Ada code, let's revisit the top-down decomposition process in the context of a "typical" computer program (at least at the introductory level). You may be surprised to realize that many, many computer programs do the same sequence of four actions:

- Describe the program (print an introduction)
- Get input from the user
- Perform calculations using that input
- Provide output to the user

By the time we get to the end of this book, we'll address some slightly more complicated programs, but for now most of the programs you write will accomplish the above four actions. That means that you'll end up picking subproblems (and then procedures) to print an introduction, to get user input, to perform calculations, and to provide output. Your main program will mostly consist of calls to those procedures (it won't do much "real work" anymore). Let's look at an example:

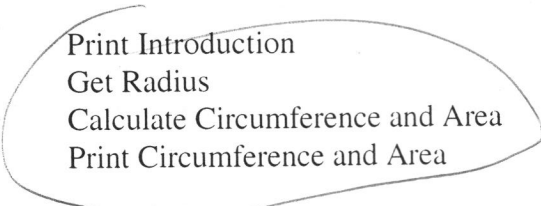
Example 7.1. Calculate Circumference and Area of a Circle

Problem Description: Write a program that will calculate the circumference and area of a circle

To start, let's do our top-down decomposition (pick our subproblems). One way to do this is to write our algorithm for the main program:

```
-- Print an introduction
-- Get the circle's radius
-- Calculate circumference and area
-- Print circumference and area
```

Notice that this algorithm isn't quite as detailed as the ones we've written before. For example, we didn't include the equations for circumference and area, and we made the calculation one step instead of the required two steps. Once you start using procedures, your main program algorithm becomes a more high-level view of how you're going to solve the problem, and you save the low-level details for the algorithms you write for each procedure. For this problem, it looks like we should break the problem into the following subproblems:



- Print Introduction
- Get Radius
- Calculate Circumference and Area
- Print Circumference and Area

We'll keep working on the solution to this problem throughout this chapter. Let's move on to the next section.

7.2. Figuring Out Information Flow

Now that we've decided which procedures we're going to write, we need to decide what information flows in to and out of each procedure. This really isn't as hard as it sounds! The only trick is to remember that we're talking about information flow between the procedure and whoever calls the procedure, NOT input from a keyboard or output to the screen. Let's look at each of the procedures from our example.

Does the procedure that prints the introduction need any information coming in? Not really, since all it does is print the introduction to the screen. Does this procedure need to pass any information out before terminating (finishing executing)? The answer is no again. So there isn't any information flow in to or out of the procedure.

How about the procedure that gets the radius? The caller doesn't provide any information to the procedure, so there's no information flow in

to the procedure (remember, this isn't about keyboard inputs). After this procedure goes to all the trouble of getting the radius, it should probably pass that radius back to the caller before terminating. So for this procedure, we have information flow (the radius) out of the procedure only.

What information does the procedure that calculates the circumference and area need and provide? To do the calculations, the procedure needs the radius of the circle, so this information flows in to the procedure. After performing the calculations, the procedure needs to pass the results back to the caller before terminating. So for this procedure, we have information flow both in to (the radius) and out of (the circumference and area) the procedure.

Finally, we have the procedure that prints the circumference and area. To be able to print these values, the procedure has to get them from somewhere, so they get passed in to the procedure. After printing, the procedure doesn't have to pass anything out to the caller (remember, this isn't about output to the screen), so there's no information flow out of the procedure. There's only information flow in to (the circumference and area) this procedure.

After picking our subproblems and figuring out our information flow, we can draw our decomposition diagram (see the following page).

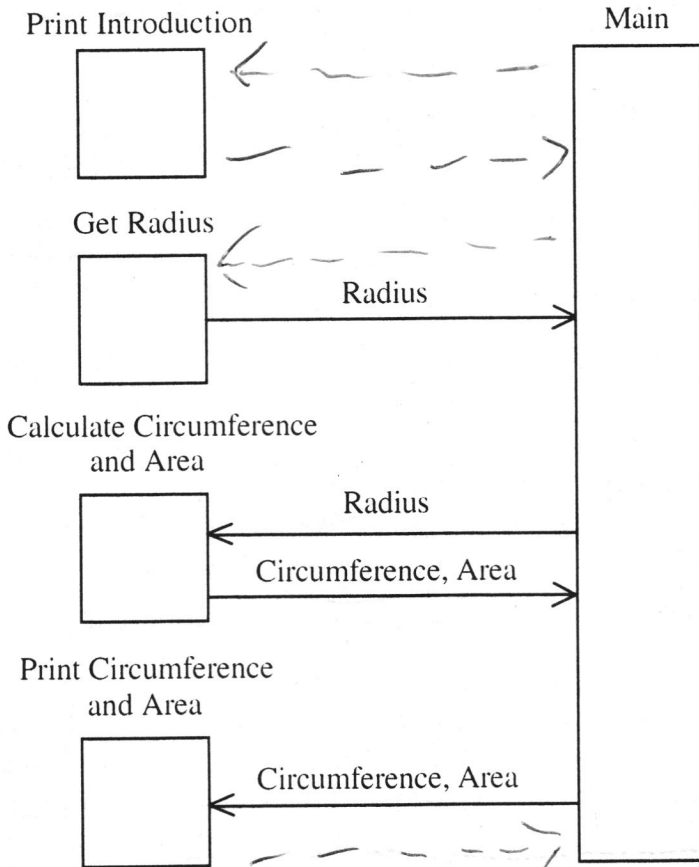
Get it? We look at a procedure, considering what it does, what it needs to do the job, and what the results are, and use this to decide what information flows in to and out of that procedure. It takes some thought, but it's really not that complicated.

7.3. Creating the Procedure Header

Now that we've decided what procedures to use and what information flows in to and out of each one, we're ready to develop our *procedure headers*. A procedure header is simply the part of our procedure that gives the name of the procedure and details about information flow in to and out of the procedure.

The procedure header for a procedure without any information flow is pretty straightforward -- here's the header for the procedure that prints the introduction (we've called it `Print_Introduction`):

procedure Print_Introduction is



Procedures that do have information flow require more complicated procedure headers. For example, the procedure that gets the radius from the user will have a procedure header that looks like:

```
procedure Get_Radius ( Radius : out Float ) is
```

Because this procedure passes information out to the caller, we use something called a parameter (Radius). Whenever we have information flow between a procedure and its caller, we use a parameter for each piece

of information. We call the parameter in the procedure header (and the procedure itself) the *formal parameter*.

The required syntax for procedure headers is as follows:

Procedure Headers

No Information Flow (No Parameters)

```
procedure <procedure name> is
```

<procedure name> - the name of the procedure.

With Information Flow (With Parameters)

```
procedure <procedure name> (  
    <formal parameter name> : <mode> <data type>;  
    <formal parameter name> : <mode> <data type>;  
    . . . ) is
```

<procedure name> - the name of the procedure.

<formal parameter name> - the name of a parameter.

<mode> - the mode (in, out, or in out) of the parameter.

<data type> - the data type for the parameter.

Each parameter has a name, a mode, and a data type. The name and data type for a parameter shouldn't really require explanation, since they're very similar to the names and data types of variables and constants. The mode, however, merits some additional discussion. When the procedure simply needs to pass information out to the caller using a parameter, the mode for that parameter should be out (surprised?). When the procedure needs the caller to pass information in to the procedure using a parameter, the mode for that parameter should be in. Finally, in some cases the procedure needs the information to come in from the caller, the procedure changes that information somehow, and then passes the information back out to the caller. In this case, the parameter mode should be in out (since the information comes both in to and out of the procedure). If you've done a good job evaluating the information flow for each procedure, picking names, modes, and data types for your parameters should be a snap.

OK, let's create the procedure headers for the last two procedures:

```

procedure Calculate_Circumference_And_Area (
    Radius          : in    Float;
    Circumference   : out Float;
    Area            : out Float ) is

```

```

procedure Print_Circumference_And_Area (
    Circumference : in Float;
    Area          : in Float ) is

```

Now that we have the procedure headers figured out, let's work on the rest of the procedures.

7.4. The Rest of the Procedure

So far, we've really only defined the "interface" for each of our procedures; we still need to write the rest of the code for each procedure. Before we do that, though, we need to write an algorithm for each one (so we know how to solve the subproblem). Since it's nice to have a comment block before each procedure to describe that procedure and to provide the algorithm, we'll create comment blocks, then complete the code, for each procedure.

```

-----
--
-- Name : Print_Introduction
-- Description : This procedure prints the introduction
--               for the program
-- Algorithm :
--   Print the introduction
--
-----

```

```

procedure Print_Introduction is
begin

```

```

    --   Print the introduction
    Put (Item => "This program calculates the ");
    Put (Item => "circumference and area");
    New_Line;
    Put (Item => "of a circle.");

```

```

    New_Line;
    New_Line;

    end Print_Introduction;

```

When the `Print_Introduction` procedure runs (we'll talk about how to make that happen in the next section), it will print the introduction to the screen. For the `Get_Radius` procedure, we have:

```

-----
--
-- Name : Get_Radius
-- Description : This procedure prompts for and gets the
--               radius of the circle
-- Algorithm :
--               Prompt for and get radius
--
-----

procedure Get_Radius ( Radius : out Float ) is
begin

    --       Prompt for and get radius
    Put (Item => "Please enter the circle's radius : ");
    Get (Item => Radius);
    Skip_Line;

end Get_Radius;

```

When this procedure runs, it will ask the user for the radius, read in that radius from the keyboard, then pass the radius out to the caller. Next we have:

```

-----
--
-- Name : Calculate_Circumference_And_Area
-- Description : This procedure uses the circle's radius
--               to calculate the circumference and area
-- Algorithm:
--               Calculate Circumference as 2 * Pi * Radius
--               Calculate Area as Pi * Radius ** 2

```

Time : in Float;
 Av. Speed : out Float;

```

-----
Get Time
procedure Calculate_Circumference_And_Area (time : out Float is
  Radius      : in      Float;
  Circumference : out Float;
  Area        : out Float ) is
    Pi : constant Float := 3.1415;

begin
  -- Calculate Circumference as 2 * Pi * Radius
  Circumference := 2.0 * Pi * Radius;

  -- Calculate Area as Pi * Radius ** 2
  Area := Pi * Radius ** 2;

end Calculate_Circumference_And_Area;
  
```

Get (Enter =) time;
~~end ()~~
 end Get Time

The only tricky thing about this procedure is that we've declared a "local" constant. When we have variables or constants that we only need in a particular procedure (not in other procedures or the main program), we declare those variables and constants between the procedure header and the begin for that procedure. These are called local variables (and constants) because they can only be used inside the procedure in which they're declared. Because this is the only procedure that needs the value of Pi, we declared the constant local to the procedure. OK, let's do the last procedure:

```

-----
--
-- Name : Print_Circumference_And_Area
-- Description : This procedure prints the circumference
--               and area of the circle
-- Algorithm :
--   Print the circumference
--   Print the area
--
-----
  
```

```

procedure Print_Circumference_And_Area (
    Circumference : in Float;
    Area           : in Float ) is
begin
    --    Print the circumference
    Put (Item => "Circumference : ");
    Put (Item => Circumference,
        Fore => 1,
        Aft  => 2,
        Exp  => 0);
    New_Line;

    --    Print the area
    Put (Item => "Area           : ");
    Put (Item => Area,
        Fore => 1,
        Aft  => 2,
        Exp  => 0);
    New_Line;

end Print_Circumference_And_Area;

```

That does it -- our procedures are done!

7.5. Calling the Procedure

Now that we have our procedures written, we need to figure out how to make them run during execution of the main program. We do this by *calling* the procedure when we want it to run. How do we call a procedure? Almost the same way your folks used to call you to dinner -- we use its name! We said "almost" the same way because we also have to worry about the parameters. Let's look at calling procedures in more detail; here's the main program code that calls the procedures:

```

-----
--  START OF MAIN PROGRAM
-----

```

```

Circle_Radius      : Float;
Circle_Circumference : Float;
Circle_Area        : Float;

begin

    --    Print introduction
    Print_Introduction;

    --    Get radius
    Get_Radius (Radius => Circle_Radius);

    --    Calculate circumference and area
    Calculate_Circumference_And_Area (
        Radius      => Circle_Radius,
        Circumference => Circle_Circumference,
        Area        => Circle_Area );

    --    Print circumference and area
    Print_Circumference_And_Area (
        Circumference => Circle_Circumference,
        Area          => Circle_Area );

end Circle_Stuff;

```

Calling `Print_Introduction` was easy -- we just put the procedure name, because that procedure doesn't have any parameters. It's a little more complicated for calling procedures with parameters, because for each formal parameter in the procedure header, we need to provide an *actual parameter* in the procedure call. The syntax for calling procedures is on the next page.

Let's consider the call to `Get_Radius`:

```
Get_Radius (Radius => Circle_Radius);
```

To call the procedure, we put the procedure name, then for each formal parameter in the procedure header we provide an actual parameter in the procedure call. Because `Get_Radius` only has one formal parameter (`Radius`), we only need to provide one actual parameter (`Circle_Radius`). In our example, the names of the formal and actual

parameters are different, but Ada will also let you use the same name for both if you so desire.

There are, of course, some rules we need to follow for parameters. If the procedure we're calling has one formal parameter, we can't call that procedure with no actual parameters, two actual parameters, etc. -- we have to call that procedure with exactly one actual parameter. In other words, the *number* of formal and actual parameters has to match. For each parameter in the call, the *data type* of the actual parameter has to match the data type of the formal parameter (actually, they only have to be compatible, but for our purposes we'll say they have to be the same). For example, the formal parameter `Radius` in `Get_Radius` is declared as a `Float`, so the actual parameter in the procedure call (`Circle_Radius`) must also be declared as a `Float`.

Procedure Calls

No Parameters

```
<procedure name>;
```

<procedure name> - the name of the procedure.

With Parameters

```
<procedure name> (  
    <formal parameter name> => <actual parameter name>,  
    <formal parameter name> => <actual parameter name>,  
    . . . );
```

<procedure name> - the name of the procedure.

<formal parameter name> - the name of a formal parameter.

<actual parameter name> - the name of an actual parameter.

We manage to avoid one other potential problem in the procedure call by using *named association* (putting the formal parameter name, that cool arrow thing, then the actual parameter name for each parameter). Ada doesn't actually require named association, though. For example, we could make our call to `Calculate_Circumference_And_Area` look like this:

```
Calculate_Circumference_And_Area (Circle_Radius,
    Circle_Circumference,
    Circle_Area );
```

and our program will work exactly the same way. But what if we make a mistake and swap the last two actual parameters, using:

```
Calculate_Circumference_And_Area (Circle_Radius,
    Circle_Area,
    Circle_Circumference );
```

instead? The program will still compile and run (the number and data type requirements for the parameters are still met), but it will get the wrong answer, putting the circumference into `Circle_Area` and the area into `Circle_Circumference`! These types of errors can be VERY hard to find. When we don't use named association, Ada uses the *order* in which we've listed our actual parameters in the procedure call to match up the formal and actual parameters. Because this can lead to hard-to-find errors for beginning (and experienced) programmers, we use named association in all our procedure calls.

As we've said, calling a procedure makes the code in that procedure run. Let's look at our entire program and see what happens from start to finish. Our programs always start right after the begin for the main program, so the first thing that executes is the call to `Print_Introduction`. The program goes to that procedure, executes each line in the procedure, then returns to the next line in the main program. The next line is the call to `Get_Radius`, so the program goes to that procedure, executes each line in the procedure, and returns to the next line in the main program. The remaining procedures are executed in the same way, then the program stops.

Do you see the common pattern we mentioned earlier in this chapter in the programs we've written so far (whether or not we use procedures)? We typically describe what the program does, get some input from the user (one or more values), perform some calculations, then provide output to the user (again, of one or more values). Not every program will follow this pattern, but most of the programs you write in an introductory course will.

Parameters seem to be a very difficult concept for programmers learning about procedures to really nail down, so the next section takes a closer look at how they work.

7.6. Parameters and How They Work

Perhaps the easiest way to understand how parameters work is to think of them this way: whenever we associate an actual parameter with a formal parameter (in the procedure call), that actual parameter is temporarily renamed to the formal parameter name. Let's illustrate this idea by modifying our program above to simply read in 2 radii. Here's the resulting main program portion:

```
-----
-- START OF MAIN PROGRAM
-----

Circle_Radius_1      : Float;
Circle_Radius_2      : Float;

begin

    --      Get first radius
    Get_Radius (Radius => Circle_Radius_1);

    --      Get second radius
    Get_Radius (Radius => Circle_Radius_2);

end Circle_Stuff;
```

What are the values of `Circle_Radius_1` and `Circle_Radius_2` before the first call to `Get_Radius`? Since we didn't initialize the variables, their values are unknown (garbage). Now, when we call `Get_Radius` the first time, any time `Radius` is referenced inside the procedure, we're really talking about the `Circle_Radius_1` variable. Why? Because in the procedure call, we said the formal parameter `Radius` corresponds to the actual parameter `Circle_Radius_1`. After the procedure finishes, `Circle_Radius_1` will have whatever value the user entered, and `Circle_Radius_2` will still have an unknown value. Now we call the

Get_Radius procedure again, this time associating Radius with Circle_Radius_2. Any time Radius is referenced inside the procedure this time, we're really talking about the Circle_Radius_2 variable. After the procedure completes this time, Circle_Radius_2 will contain whatever data value the user entered second.

We write procedures to do general sorts of things (like reading in a radius). We can then reuse these procedures as many times as we want, and by using different actual parameters each time, we can make those procedures act on as many different variables as we want. The ability to use the same subproblem solution (procedure) many times is one of the things that makes procedures so useful, and parameters are an essential part of making this reuse possible.

7.7. Putting It All Together

Now let's go through the entire problem-solving process for a problem that needs to use (or at least should use) procedures. Here's the problem description:

Read in the user's points (0.0, 1.0, 2.0, 3.0, or 4.0) for 5 courses, all worth 3 semester hours. Calculate and print the user's overall GPA.

Understand the Problem

Do we understand the problem? This one is a little confusing. For instance, why aren't we having the user enter their letter grade for each class instead of having them enter the points for each class? The real answer is because you don't yet know how to convert letter grades to points (you'll learn how to do that in the next chapter). So for now we'll just do what the problem description says, using the assumption that an A is worth 4.0 points, a B is worth 3.0 points, and so on. What if the user enters an invalid point value? We'll accept it (and even use it in the overall GPA calculation), knowing that in a couple chapters we'll learn how to handle invalid input. Let's move on to the next step.

Design a Solution

First we'll use top-down decomposition to break the problem into smaller subproblems, then we'll write detailed algorithms to solve each of the subproblems.

To help us decide what the subproblems should be, let's write a high-level algorithm for our solution:

```
-- Print introduction
-- Get points for first course
-- Get points for second course
-- Get points for third course
-- Get points for fourth course
-- Get points for fifth course
-- Calculate GPA
-- Print out GPA
```

All of these steps look like good candidates for subproblems (though of course we'll only need to figure out how to read in the score once). Specifically, let's break the problem into the following subproblems:

```
Print Introduction
Get Course Points
Calculate GPA
Print GPA
```

Since we know that we'll implement each of these subproblems as a procedure, let's write the comment blocks (including the detailed algorithms) for each of the procedures:

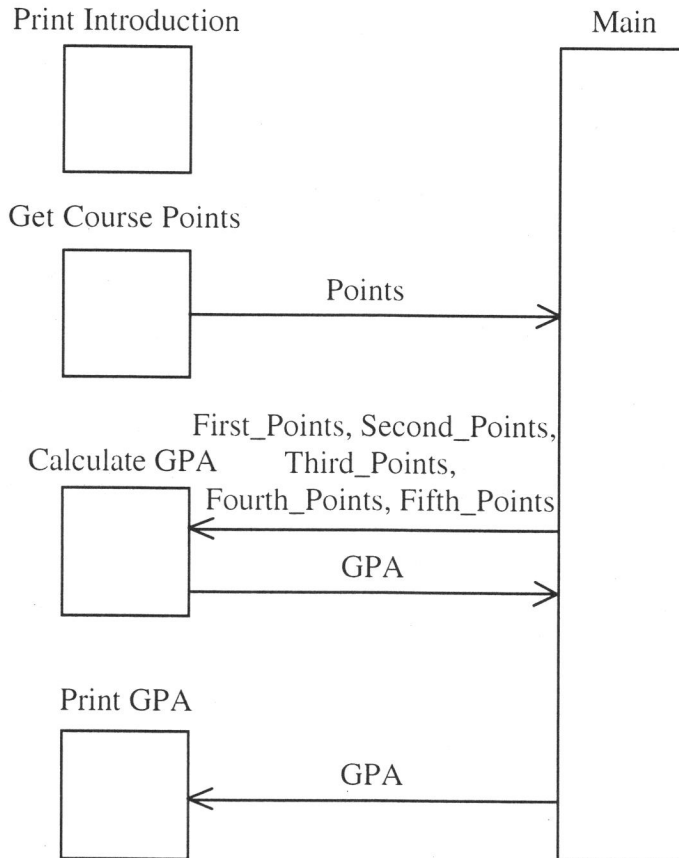
```
-----
--
-- Name : Print_Introduction
-- Description : This procedure prints the introduction for
--               the program
-- Algorithm :
--               Print the introduction
--
-----
```

```

-----
--
-- Name : Get_Course_Points
-- Description : This procedure gets the points for one
--               course from the user
-- Algorithm :
--               Prompt for and get points
--
-----
--
-- Name : Calculate_GPA
-- Description : This procedure calculates the GPA for the
--               given course points
-- Algorithm :
--               Set Sum to the sum of all the points
--               Calculate GPA as Sum / 5.0
--
-----
--
-- Name : Print_GPA
-- Description : This procedure prints the GPA
-- Algorithm :
--               Print the GPA
--
-----

```

Before we move on to the next step, there's one more thing to do in the design - decide what information needs to go in to and out of each procedure. The `Print_Introduction` procedure doesn't need any parameters. Because the `Get_Course_Points` procedure gets the course points from the user, it should pass those points out to the main program. `Calculate_GPA` needs the points to come in to the procedure, then should pass the GPA out to the main program. `Print_GPA` needs the GPA to print it, so the GPA should get passed in to this procedure from the main program. These information flow choices yield the decomposition diagram on the following page.



Let's write the procedure headers for each of these procedures so we remember what we've decided for the information flow:

```
procedure Print_Introduction is
```

```
procedure Get_Course_Points ( Points : out Float ) is
```

```
procedure Calculate_GPA ( First_Points : in      Float;
    Second_Points : in      Float;
    Third_Points  : in      Float;
    Fourth_Points : in      Float;
    Fifth_Points  : in      Float;
    GPA           :      out Float ) is
```

```
procedure Print_GPA ( GPA : in Float ) is
```

Now we're ready to move on to our test plan.

Write the Test Plan

Because this program doesn't contain any selection or iteration constructs, we can use a single test case for our test plan.

Test Case 1 : Checking GPA Calculation

Step	Input	Expected Results	Actual Results
1	0.0 for first points	Prompt for second points	
2	1.0 for second points	Prompt for third points	
3	2.0 for third points	Prompt for fourth points	
4	3.0 for fourth points	Prompt for fifth points	
5	4.0 for fifth points	Print 2.00 for GPA	

Write the Code

Let's write the code from our algorithms; here's what it looks like:

★ *USE THIS AS Example*

```
--
-- Author : Ted Nugent
-- Description : This program reads in the user's points
--               (0.0, 1.0, 2.0, 3.0, or 4.0) for 5 courses, all worth
--               3 semester hours. It then calculates and prints the
--               user's overall GPA.
-- Algorithm :
--   Print introduction
--   Get points for first course
--   Get points for second course
--   Get points for third course
--   Get points for fourth course
```



```
--      Get points for fifth course
--      Calculate GPA
--      Print out GPA
--
```

```
with Ada.Text_IO;
use  Ada.Text_IO;
```

```
with Ada.Float_Text_IO;
use  Ada.Float_Text_IO;
```

```
procedure Calculate_GPA is
```

```
--
-- Name : Print_Introduction
-- Description : This procedure prints the introduction
--               for the program
-- Algorithm :
--               Print the introduction
--
```

```
procedure Print_Introduction is
begin
```

```
--      Print the introduction
Put (Item => "This program reads in the user's ");
Put (Item => "points (0.0, 1.0, 2.0, 3.0, ");
New_Line;
Put (Item => "or 4.0) for 5 courses, all worth ");
Put (Item => "3 semester hours.  It then ");
New_Line;
Put (Item => "calculates and prints the user's ");
Put (Item => "overall GPA.");
New_Line;
New_Line;
```

```
end Print_Introduction;
```

```

-----
--
-- Name : Get_Course_Points
-- Description : This procedure gets the points for one
--               course from the user
-- Algorithm :
--               Prompt for and get points
--
-----

```

```

procedure Get_Course_Points ( Points : out Float ) is
begin

```

```

    -- Prompt for and get points
    Put (Item => "Please enter course points ");
    Put (Item => "(0, 1, 2, 3, or 4) : ");
    Get (Item => Points );
    Skip_Line;

```

```

end Get_Course_Points;

```

```

-----
--
-- Name : Calculate_GPA
-- Description : This procedure calculates the GPA for
--               the given course points
-- Algorithm :
--               Set Sum to the sum of all the points
--               Calculate GPA as Sum / 5.0
--
-----

```

```

procedure Calculate_GPA ( First_Points : in      Float;
                          Second_Points : in     Float;
                          Third_Points  : in     Float;
                          Fourth_Points : in     Float;
                          Fifth_Points  : in     Float;
                          GPA           :      out Float ) is

    Sum : Float;

```

```

begin

```

```

--      Set Sum to the sum of all the points
Sum := First_Points + Second_Points + Third_Points +
      Fourth_Points + Fifth_Points;

--      Calculate GPA as Sum / 5.0
GPA := Sum / 5.0;

end Calculate_GPA;

-----
--
-- Name : Print_GPA
-- Description : This procedure prints the GPA
-- Algorithm :
--      Print the GPA
--
-----

procedure Print_GPA ( GPA : in Float ) is
begin
    --      Print the GPA
    Put (Item => "Overall GPA : ");
    Put (Item => GPA,
        Fore => 1,
        Aft  => 2,
        Exp  => 0);
    New_Line;

end Print_GPA;

-----
-- START OF MAIN PROGRAM
-----

Points_1 : Float;
Points_2 : Float;
Points_3 : Float;
Points_4 : Float;
Points_5 : Float;
GPA      : Float;

```

```

begin
    --      Print introduction
    Print_Introduction;

    --      Get points for first course
    Get_Course_Points ( Points => Points_1 );

    --      Get points for second course
    Get_Course_Points ( Points => Points_2 );

    --      Get points for third course
    Get_Course_Points ( Points => Points_3 );

    --      Get points for fourth course
    Get_Course_Points ( Points => Points_4 );

    --      Get points for fifth course
    Get_Course_Points ( Points => Points_5 );

    --      Calculate GPA
    Calculate_GPA ( First_Points => Points_1,
        Second_Points => Points_2,
        Third_Points  => Points_3,
        Fourth_Points => Points_4,
        Fifth_Points  => Points_5,
        GPA           => GPA );

    --      Print out GPA
    Print_GPA ( GPA => GPA );

end Calculate_GPA;

```

Test the Code

So now we need to run our test plan to make sure the program works. When we run the program above and fill in the actual results in our test plan, we get the following:

Test Case 1 : Checking GPA Calculation

Step	Input	Expected Results	Actual Results
1	0.0 for first points	Prompt for second points	Prompts for second points
2	1.0 for second points	Prompt for third points	Prompts for third points
3	2.0 for third points	Prompt for fourth points	Prompts for fourth points
4	3.0 for fourth points	Prompt for fifth points	Prompts for fifth points
5	4.0 for fifth points	Print 2.00 for GPA	Prints 2.00 for GPA

7.8. Common Mistakes*Not Including an Actual Parameter for Each Formal Parameter*

When we write our procedure header, we define the interface to that procedure. Any code that calls this procedure has to follow the interface, providing exactly one actual parameter for each formal parameter in the procedure header. Trying to call the procedure with too few or too many actual parameters results in a compilation error.

Mismatched Data Types Between Actual and Formal Parameters

This is again an interface issue. The data type of an actual parameter must match (be compatible with, to be more precise, but we won't worry about that distinction here) the data type of the formal parameter for which it's being provided. A type mismatch between the actual and formal parameters results in a compilation error.

Using Incorrect Parameter Modes

Yet another interface issue. When we decide on our information flow during our top-down decomposition, we determine in which direction each piece of information flows. If the information flows out of the procedure to the caller, for example, the mode for that parameter should be `out`. Remember, the parameter mode has NOTHING to do with any input and

output devices you might be using (keyboard, screen, etc.), so a procedure like `Get_Course_Points` above passes `Points` *out* of the procedure, even though the user uses an input device to enter them.

Procedure Never Runs

Remember, to make a procedure run, you actually have to call it from the main program (or from another procedure). If you've written a procedure that never runs when you execute your program, you probably forgot to add the code that calls the procedure.

7.9. Exercises

1. Write a program that reads in 3 GPAs, then prints each GPA. Do NOT print each GPA as it's read; rather, read in all 3, then print them.
2. Read in 2 weights (as Floats), 2 dollar amounts, and 2 test percentages. Calculate the average for each, then print out the averages.
3. Read in the radius of a circle and the side length of a square. Calculate the circumference and area for both. Print out the input and calculated values.

Chapter 8. Selection

Well, we've talked about how we can solve problems by breaking them into smaller subproblems, then solving each of those subproblems using procedures. Have you noticed, though, that each of our subproblems have been fairly easy to solve? The programs haven't even had to choose between different steps to execute -- they've simply started at the beginning and gone until they're done.

While these kinds of problems helped us concentrate on understanding procedures without lots of other distractions, it's time to consider some more complicated problems. Specifically, this chapter discusses some of the ways we can set up our program to choose, or select, between different statements to execute. Let's get to it.

8.1. If Statements

One construct we can use to choose between different courses of action is the `if` statement. There are actually a number of different ways we can use this statement; the syntax for the most basic way is provided on the following page.

The simplest use of the `if` statement is the *one-alternative if statement*. We use this form of the statement when we simply need to decide whether or not to perform a particular action. For example, say you needed to decide if a student is on the Dean's List (on the list of students with GPAs of 3.0 or higher). Here's one algorithm you could use:

```
-- If GPA greater than or equal to 3.0
-- Print Dean's List message
```

and turning this into code, we get:

```
-- If GPA greater than or equal to 3.0
if ( GPA >= 3.0 ) then

    -- Print Dean's List message
    Put (Item => "You made Dean's List !!");
```

```
New_Line;

end if;
```

Here's the syntax for the one-alternative `if` statement:

One-Alternative If Statement

```
if <Boolean expression> then
    <executable statements>
end if;
```

<Boolean expression> - some expression that evaluates to True or False.
 <executable statements> - one or more executable statements (like assignment statements, procedure calls, etc.)

We put a Boolean expression between the `if` and the `then`. Recall from Chapter 3 that a Boolean expression is an expression that evaluates to either True or False.

We actually snuck something else new into our discussion (the `>=`), though we did include it in the table of operators in Chapter 6. There are a number of *relational operators* in Ada that let us compare two things (try to figure out the relationship between them). Those operators are:

<code>=</code>	equal to
<code>/=</code>	not equal to
<code><</code>	less than
<code><=</code>	less than or equal to
<code>></code>	greater than
<code>>=</code>	greater than or equal to

The Ada operators (including relational operators) have an *order of precedence*. From your math courses, you know that in a math equation we evaluate multiplications before additions because multiplication has a higher order of precedence. Here are all the operators in Ada, listed from

highest to lowest precedence (operators on the same line have the same precedence):

```

** (exponentiation), abs (absolute value), not
*, /, mod (modulus), rem (remainder)
+, - (unary operators)
+, -, & (binary operators)
=, /=, <, <=, >, >=
and, or, xor (exclusive-or)

```

But how does a one-alternative `if` statement actually use the Boolean expression we write? When the program gets to the `if`, it evaluates the Boolean expression. If the Boolean expression evaluates to `True`, the statements between the `then` and the `end if` are executed; otherwise, the program just skips to the line in the program after the `end if`. See how we use the Boolean expression to decide (or select) which code to execute?

The *two-alternative if statement* takes us one step further by letting us choose between two alternatives. The syntax for the two-alternative `if` statement is as follows:

Two-Alternative If Statement

```

if <Boolean expression> then
    <executable statements>
else
    <executable statements>
end if;

```

<Boolean expression> - some expression that evaluates to `True` or `False`.
 <executable statements> - one or more executable statements.

Let's extend our example to print a sympathetic message if the student hasn't made Dean's List. Our algorithm becomes:

```
-- If GPA greater than or equal to 3.0
-- Print Dean's List message
-- Otherwise
-- Print sympathetic message
```

and the resulting code:

```
-- If GPA greater than or equal to 3.0
if ( GPA >= 3.0 ) then

    -- Print Dean's List message
    Put (Item => "You made Dean's List !!");
    New_Line;

-- Otherwise
else

    -- Print sympathetic message
    Put (Item => "Not on Dean's List.  Better ");
    Put (Item => "luck next time.");
    New_Line;

end if;
```

The program still evaluates the Boolean expression when it gets to the `if`. If it evaluates to `True`, the program executes the statements between the `if` and the `else`, then skips to the line in the program after the `end if`. If the Boolean expression evaluates to `False`, the program executes the statements between the `else` and the `end if`. Since the Boolean expression can only evaluate to `True` or `False`, either the `if` portion or the `else` portion is always executed in a two-alternative `if` statement.

What if we have multiple (more than two) alternatives we'd like to select from? We use the *multiple-alternative if statement* (what a shocker). The syntax for the multiple-alternative `if` statement is on the following page.

Let's expand our algorithm just a bit more to print the sympathetic message if the student's GPA is greater than or equal to 2.0 (and less than 3.0), and inform them that they're on academic probation if their GPA is less than 2.0. Here's the algorithm:

```
-- If GPA greater than or equal to 3.0
  -- Print Dean's List message
-- Otherwise, if GPA greater than or equal to 2.0
  -- Print sympathetic message
-- Otherwise
  -- Print academic probation message
```

Multiple-Alternative If Statement

```
if <Boolean expression> then
    <executable statements>
elseif <Boolean expression> then
    <executable statements>
elseif <Boolean expression> then
    <executable statements>
else
    <executable statements>
end if;
```

<Boolean expression> - some expression that evaluates to true or false.
 <executable statements> - one or more executable statements.

and the code for our algorithm is:

```
-- If GPA greater than or equal to 3.0
if ( GPA >= 3.0 ) then

    -- Print Dean's List message
    Put (Item => "You made Dean's List !!");
    New_Line;
```

```

-- Otherwise, if GPA greater than or equal to 2.0
elsif ( GPA >= 2.0 ) then

    -- Print sympathetic message
    Put (Item => "Not on Dean's List.  Better ");
    Put (Item => "luck next time.");
    New_Line;

-- Otherwise
else

    -- Print academic probation message
    Put (Item => "You're on Academic Probation !!");
    New_Line;

end if;

```

The program evaluates the first Boolean expression when it gets to the `if`; if it evaluates to `True`, the program executes the statements between the `if` and the `elsif`, then skips to the line in the program after the `end if`. If the Boolean expression evaluates to `False`, the program goes to the `elsif` and evaluates the Boolean expression there; if it evaluates to `True`, the program executes the statements between the `elsif` and the `else`, then skips to the line in the program after the `end if`. Finally, if none of the preceding alternatives have been selected, the program executes the `else` portion of the `if` statement. The easiest way to remember how this works is to remember that the program goes "from the top down" -- it will execute the statements in the **FIRST** alternative for which the Boolean expression is `True` (or the `else` if it gets all the way there), then skips to the program line after the `end if`. Multiple-alternative `if` statements can have as many `elsif` portions as you need (each of which has its own Boolean expression, of course), and the `else` portion in a multiple-alternative `if` statement is optional.

Given the rule that the statements are executed in the first alternative for which the Boolean expression evaluates to `True`, can you see why we didn't need:

```

elsif ( GPA >= 2.0 ) and ( GPA < 3.0 ) then

```

in the example above? If GPA is greater than or equal to 3.0, the first alternative in the `if` statement would have been executed, so the only way we can even get to this `elsif` is if the GPA is less than 3.0. Including the check for GPA less than 3.0 in the `elsif` doesn't change the way the code works (because the `GPA < 3.0` part always evaluates to True), but it does make the code more complicated than it needs to be.

We should point out that the statements contained inside an `if` statement can also be `if` statements; we call these *nested* `if` statements. We'll see an example of nested `if` statements in Section 8.3.

So now you know how to add selection to your programs using the various forms of the `if` statement. Cool.

8.2. Case Statements

In some situations, a case statement provides a convenient way to select between different courses of action. Here's the syntax:

Case Statement

```
case <variable name> is  
    when <list of values> => <executable statements>  
    when <list of values> => <executable statements>  
    .  
    .  
    when others => <executable statements>  
end case;
```

<variable name> - the name of a variable.

<list of values> - a list of possible values of the variable.

<executable statements> - one or more executable statements.

We said case statements are appropriate "in some cases" because the data type of the variable used for the alternative selection is restricted. For our purposes, this variable can be either an integer or a character. The variable's data type is restricted so that we can explicitly list possible values of the variable after each `when`. This can be a little confusing, so let's look

at an example. Say we wanted a case statement that will print the appropriate letter grade given a test score (i.e., A for 90-100, B for 80-89, and so on). The algorithm looks like:

```
-- When score is between 90 and 100
-- Print A
-- When score is between 80 and 89
-- Print B
-- When score is between 70 and 79
-- Print C
-- When score is between 60 and 69
-- Print D
-- Otherwise
-- Print F
```

and the code (using a case statement, assuming Test_Score is an integer variable holding the test score) is:

```
case Test_Score is

    -- When score is between 90 and 100
    when 90 .. 100 =>

        -- Print A
        Put_Line (Item => "You got an A");

    -- When score is between 80 and 89
    when 80 .. 89 =>

        -- Print B
        Put_Line (Item => "You got a B");

    -- When score is between 70 and 79
    when 70 .. 79 =>

        -- Print C
        Put_Line (Item => "You got a C");

    -- When score is between 60 and 69
    when 60 .. 69 =>
```

```

-- Print D
Put_Line (Item => "You got a D");

-- Otherwise
when others =>

    -- Print F
    Put_Line (Item => "You got an F");

end case;

```

When we put 90 .. 100 after the `when`, that tells the program to select that alternative if `Test_Score` is 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, or 100. Using the .. notation let us simply put the lower and upper bounds on the values, but we could also have listed each value explicitly using the `|` notation as follows, where `|` means "or":

```

when 90 | 91 | 92 | 93 | 94 |
      95 | 96 | 97 | 98 | 99 | 100 =>

```

You should also know that EVERY possible value of `Test_Score` has to be covered in exactly one of the lists of values after a `when` in the case statement. Using the `when others` at the end of the case statement (the only place you're allowed to put it, by the way), is a shorthand way of saying "all the other possible values not listed above". Each value can only be listed in one place, though; otherwise, the program wouldn't know which alternative to select for that value.

So when the computer gets to the case statement, it figures out the current value of the variable, executes the alternative with that value listed after the `when`, then skips to the line in the program after the `end case`. Case statements therefore give us another way to select between alternatives in our programs.

8.3. Putting It All Together

Now let's go through the entire problem-solving process for a problem that requires selection. Here's the problem description:

Read in three integers from the user. Print out the whether they're all the same, two of them are the same, or none of them are the same.

Understand the Problem

Do we understand the problem? WHAT we need to do seems pretty clear, even if we don't quite know HOW to do it yet. Our only real question would be "If two of the numbers are the same, do we have to say WHICH two are the same?" For this problem, we won't require that, so let's move on to the next step.

Design a Solution

First let's write a high-level algorithm for our solution:

```
-- Print introduction
-- Get first number from user
-- Get second number from user
-- Get third number from user
-- Determine how many are the same
-- Print out message
```

All of these steps look like good candidates for subproblems (though of course we'll only need to figure out how to get a number from the user once). Specifically, let's break the problem into the following subproblems:

```
Print Introduction
Get Number
Determine Same Count
Print Message
```

Next, we write the comment blocks (including the detailed algorithms) for each of the procedures:

```
-----
--
-- Name : Print_Introduction
-- Description : This procedure prints the introduction for
--               the program
```



```

-- Algorithm :
--   Print the introduction
--
-----

--
-----
-- Name : Get_Number
-- Description : This procedure gets a number from the
--   user
-- Algorithm :
--   Prompt for and get number
--
-----

-----
--
-----
-- Name : Determine_Same_Count
-- Description : This procedure determines how many of
--   the three numbers are the same
-- Algorithm :
--   If first number = second number
--     If first number = third number
--       Set Same_Count to 3
--     Otherwise
--       Set Same_Count to 2
--   Otherwise, if first number = third number
--     Set Same_Count to 2
--   Otherwise, if second number = third number
--     Set Same_Count to 2
--   Otherwise,
--     Set Same_Count to 0
--
-----

-----
--
-----
-- Name : Print_Message
-- Description : This procedure prints how many of the
--   numbers are the same

```

```
-- Algorithm :
--   Print message
--
```

The `Print_Introduction` procedure doesn't need any parameters. Because the `Get_Number` procedure gets a number from the user, it should pass that number out to the main program. The `Determine_Same_Count` procedure needs all three numbers to come in (so it can figure out how many are the same), then needs to pass out how many are the same to the main program. `Print_Message` needs to know how many numbers were the same, so this needs to get passed in to this procedure from the main program. The resulting decomposition diagram is on the following page.

Our procedure headers are as follows:

```
procedure Print_Introduction is

procedure Get_Number ( Number : out Integer ) is

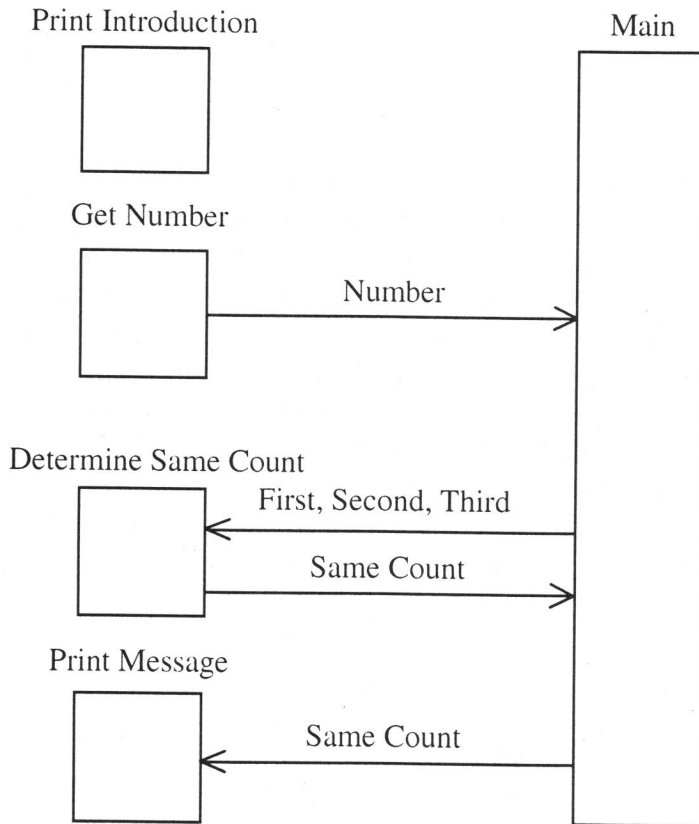
procedure Determine_Same_Count ( First : in      Integer;
    Second      : in      Integer;
    Third       : in      Integer;
    Same_Count  :      out Integer ) is

procedure Print_Message ( Same_Count : in Integer ) is
```

Now we're ready to move on to our test plan.

Write the Test Plan

So how are we going to test our code? We need to try all branches in the code; that leads to 5 test cases. We also have 3 boundary values to check for each branch. For example, since our first check is to see if `first = second`, we should have test cases with the second number equal to the first number - 1, the second number equal to the first number, and the second number equal to the first number + 1. Checking the boundary values for each branch adds three more test cases, giving us eight test cases for this problem.



Test Case 1 : Checking Right Branch (First = Second) then Right Branch (First = Third) and Boundary Values First = Second (Branch 1) and First = Third (Branch 2)

Step	Input	Expected Results	Actual Results
1	1 for first number	Prompt for second number	
2	1 for second number	Prompt for third number	
3	1 for third number	Print 3 numbers are the same	

**Test Case 2 : Checking Right Branch (First = Second) then
Left Branch (First \neq Third) and
Boundary Value First = Third - 1 (Branch 2)**

Step	Input	Expected Results	Actual Results
1	1 for first number	Prompt for second number	
2	1 for second number	Prompt for third number	
3	2 for third number	Print 2 numbers are the same	

**Test Case 3 : Checking Left Branch (First \neq Second) then
Right Branch (First = Third) and
Boundary Values First = Second + 1 (Branch 1)
and First = Third (Branch 3)**

Step	Input	Expected Results	Actual Results
1	2 for first number	Prompt for second number	
2	1 for second number	Prompt for third number	
3	2 for third number	Print 2 numbers are the same	

**Test Case 4 : Checking Left Branch (First \neq Second) then
 Left Branch (First \neq Third) then
 Right Branch (Second = Third) and
 Boundary Values First = Second - 1 (Branch 1)
 and First = Third - 1 (Branch 3) and
 Second = Third (Branch 4)**

Step	Input	Expected Results	Actual Results
1	1 for first number	Prompt for second number	
2	2 for second number	Prompt for third number	
3	2 for third number	Print 2 numbers are the same	

**Test Case 5 : Checking Left Branch (First \neq Second) then
 Left Branch (First \neq Third) then
 Left Branch (Second \neq Third) and
 Boundary Value Second = Third + 1 (Branch 4)**

Step	Input	Expected Results	Actual Results
1	3 for first number	Prompt for second number	
2	2 for second number	Prompt for third number	
3	1 for third number	Print 0 numbers are the same	

Test Case 6 : Checking Boundary Value First = Third + 1 (Branch 2)

Step	Input	Expected Results	Actual Results
1	2 for first number	Prompt for second number	
2	2 for second number	Prompt for third number	
3	1 for third number	Print 2 numbers are the same	

Test Case 7 : Checking Boundary Value First = Third + 1 (Branch 3)

Step	Input	Expected Results	Actual Results
1	2 for first number	Prompt for second number	
2	1 for second number	Prompt for third number	
3	1 for third number	Print 2 numbers are the same	

Test Case 8 : Checking Boundary Value Second = Third - 1 (Branch 4)

Step	Input	Expected Results	Actual Results
1	1 for first number	Prompt for second number	
2	2 for second number	Prompt for third number	
3	3 for third number	Print 0 numbers are the same	

Write the Code

Let's write the code from our algorithms; here's what it looks like:

```

-----
--
-- Author : Mick Jones
-- Description : This program reads in three integers from
--               the user. It then prints out whether all three are
--               the same, whether two are the same, or whether none
--               are the same.
-- Algorithm :
--   Print introduction
--   Get first number from user
--   Get second number from user
--   Get third number from user
--   Determine how many are the same
--   Print out message
--
-----

```

```

with Ada.Text_IO;
use  Ada.Text_IO;

```

```

with Ada.Integer_Text_IO;
use  Ada.Integer_Text_IO;

```

```

procedure Count_Same_Numbers is

```

```

-----
--
-- Name : Print_Introduction
-- Description : This procedure prints the introduction
--               for the program
-- Algorithm :
--   Print the introduction
--
-----

```

```

procedure Print_Introduction is
begin

```

```

    --   Print the introduction
    Put (Item => "This program reads in three integers ");
    Put (Item => "from the user. It then prints ");
    New_Line;
    Put (Item => "out whether all three are the same, ");

```

```

Put (Item => "whether two are the same, or");
New_Line;
Put (Item => "whether none are the same.");
New_Line;
New_Line;

end Print_Introduction;

-----
--
-- Name : Get_Number
-- Description : This procedure gets a number from the
--               user
-- Algorithm :
--               Prompt for and get number
--
-----

procedure Get_Number ( Number : out Integer ) is
begin
    -- Prompt for and get number
    Put (Item => "Please enter an integer : ");
    Get (Item => Number );
    Skip_Line;

end Get_Number;

-----
--
-- Name : Determine_Same_Count
-- Description : This procedure determines how many of
--               the three numbers are the same
-- Algorithm :
--               If first number = second number
--                   If first number = third number
--                       Set Same_Count to 3
--                   Otherwise
--                       Set Same_Count to 2
--               Otherwise, if first number = third number
--                   Set Same_Count to 2
--               Otherwise, if second number = third number
--                   Set Same_Count to 2
--

```



```

--      Otherwise,
--          Set Same_Count to 0
--
-----

procedure Determine_Same_Count ( First : in      Integer;
    Second      : in      Integer;
    Third       : in      Integer;
    Same_Count  :      out Integer ) is
begin

    --      If first number = second number
    if ( First = Second ) then

        --          If first number = third number
        if ( First = Third ) then

            --              Set Same_Count to 3
            Same_Count := 3;

        --      Otherwise
        else

            --              Set Same_Count to 2
            Same_Count := 2;

        end if;

    --      Otherwise, if first number = third number
    elsif ( First = Third ) then

        --          Set Same_Count to 2
        Same_Count := 2;

    --      Otherwise, if second number = third number
    elsif ( Second = Third ) then

        --          Set Same_Count to 2
        Same_Count := 2;

    --      Otherwise,
    else

```

```

--          Set Same_Count to 0
Same_Count := 0;

end if;

end Determine_Same_Count;

-----
--
-- Name : Print_Message
-- Description : This procedure prints how many of the
--               numbers are the same
-- Algorithm :
--               Print message
--
-----

procedure Print_Message ( Same_Count : in Integer ) is
begin
    --       Print message
    Put (Item => Same_Count,
        Width => 1);
    Put (Item => " numbers are the same.");
    New_Line;

end Print_Message;

-----
-- START OF MAIN PROGRAM
-----

Number_1 : Integer;
Number_2 : Integer;
Number_3 : Integer;
Same      : Integer;

begin

    --       Print introduction
    Print_Introduction;

```

```

--      Get first number from user
Get_Number ( Number => Number_1 );

--      Get second number from user
Get_Number ( Number => Number_2 );

--      Get third number from user
Get_Number ( Number => Number_3 );

--      Determine how many are the same
Determine_Same_Count ( First => Number_1,
                        Second      => Number_2,
                        Third       => Number_3,
                        Same_Count => Same );

--      Print out message
Print_Message ( Same_Count => Same );

end Count_Same_Numbers;

```

Test the Code

When we run the program above and fill in the actual results in our test plan, we get the following:

Test Case 1 : Checking Right Branch (First = Second) then Right Branch (First = Third) and Boundary Values First = Second (Branch 1) and First = Third (Branch 2)

Step	Input	Expected Results	Actual Results
1	1 for first number	Prompt for second number	Prompts for second number
2	1 for second number	Prompt for third number	Prompts for third number
3	1 for third number	Print 3 numbers are the same	Prints 3 numbers are the same

**Test Case 2 : Checking Right Branch (First = Second) then
Left Branch (First \neq Third) and
Boundary Value First = Third - 1 (Branch 2)**

Step	Input	Expected Results	Actual Results
1	1 for first number	Prompt for second number	Prompts for second number
2	1 for second number	Prompt for third number	Prompts for third number
3	2 for third number	Print 2 numbers are the same	Prints 2 numbers are the same

**Test Case 3 : Checking Left Branch (First \neq Second) then
Right Branch (First = Third) and
Boundary Values First = Second + 1 (Branch 1)
and First = Third (Branch 3)**

Step	Input	Expected Results	Actual Results
1	2 for first number	Prompt for second number	Prompts for second number
2	1 for second number	Prompt for third number	Prompts for third number
3	2 for third number	Print 2 numbers are the same	Prints 2 numbers are the same

**Test Case 4 : Checking Left Branch (First \neq Second) then
 Left Branch (First \neq Third) then
 Right Branch (Second = Third) and
 Boundary Values First = Second - 1 (Branch 1)
 and First = Third - 1 (Branch 3) and
 Second = Third (Branch 4)**

Step	Input	Expected Results	Actual Results
1	1 for first number	Prompt for second number	Prompts for second number
2	2 for second number	Prompt for third number	Prompts for third number
3	2 for third number	Print 2 numbers are the same	Prints 2 numbers are the same

**Test Case 5 : Checking Left Branch (First \neq Second) then
 Left Branch (First \neq Third) then
 Left Branch (Second \neq Third) and
 Boundary Value Second = Third + 1 (Branch 4)**

Step	Input	Expected Results	Actual Results
1	3 for first number	Prompt for second number	Prompts for second number
2	2 for second number	Prompt for third number	Prompts for third number
3	1 for third number	Print 0 numbers are the same	Prints 0 numbers are the same

Test Case 6 : Checking Boundary Value First = Third + 1 (Branch 2)

Step	Input	Expected Results	Actual Results
1	2 for first number	Prompt for second number	Prompts for second number
2	2 for second number	Prompt for third number	Prompts for third number
3	1 for third number	Print 2 numbers are the same	Prints 2 numbers are the same

Test Case 7 : Checking Boundary Value First = Third + 1 (Branch 3)

Step	Input	Expected Results	Actual Results
1	2 for first number	Prompt for second number	Prompts for second number
2	1 for second number	Prompt for third number	Prompts for third number
3	1 for third number	Print 2 numbers are the same	Prints 2 numbers are the same

Test Case 8 : Checking Boundary Value Second = Third - 1 (Branch 4)

Step	Input	Expected Results	Actual Results
1	1 for first number	Prompt for second number	Prompts for second number
2	2 for second number	Prompt for third number	Prompts for third number
3	3 for third number	Print 0 numbers are the same	Prints 0 numbers are the same

8.4. Common Mistakes

Forgetting to Put End If

Every `if` statement has to be "closed" with an `end if`. Forgetting to end the `if` statement results in a compilation error.

Using Else If Instead of Elsif

There's actually only one reserved word in Ada that's not also an English word: `elsif`. If you try to use `else if` when you really mean `elsif`, the compiler will think you're trying to start a new `if` statement within the `else` portion of the existing `if` statement, rather than simply adding an alternative to the existing `if` statement. This results in a compilation error (you won't have enough `end ifs`), but it's really an error in your logic.

Not Including All Possible Values in Case Statement

Ada requires that you include ALL possible values of the case variable in the alternatives. If there are some variable values for which you don't need to do anything, simply include

```
when others => null;
```

as the last alternative in your case statement.

8.5. Exercises

1. Write a program that reads in the radius of a circle and the side length of a square. Calculate the area for both and determine which area is larger.
2. Write a program that reads in one of 4 characters: R, B, J, or C. Based on the character entered, the program should print "Rock and Roll Rules" (for an R), "Blues Rules" (for a B), "Jazz Rules" (for a J), or "Classical Rules" (for a C).
3. Write a program that reads in 3 GPAs then determines how many of those GPAs qualify for the Dean's List (3.0 or higher GPA).

Chapter 9. Iteration

So far, we've learned how to use Ada to solve problems that require execution of a set of sequential steps and/or one or more selections. The set of problems we can solve is still somewhat limited, though, and this chapter presents the Ada constructs used to implement the last of the three basic control structures - iteration (also commonly called looping).

Iteration is useful when our program might need to complete certain actions multiple times. We may not even know how many times those actions should be completed, so we can't simply write code to complete the actions a set number of times. Instead, we put those actions inside a *loop* and set up the conditions on the loop so that the actions are completed the appropriate number of times. Let's write an algorithm for a simple example in which we know how many times the loop should execute, then we'll move on to a more complicated example.

Example 9.1. Printing Squares of the Integers from 1 to 10

Problem Description: Write a program that will print the squares of the integers from 1 to 10.

Algorithm: We could write an algorithm that contains the following steps:

```
-- Print 1 squared
-- Print 2 squared
-- Print 3 squared
. . .
```

This looks pretty awkward, though, and the algorithm would be much worse if we had to print the squares of the first 1,000 integers instead of the first 10! Instead, we'll develop an algorithm that contains iteration:

```
-- For each of the integers from 1 to 10
  -- Print the square of that integer
```

We use indentation to show what happens inside the loop, just as we used indentation to show what happened inside our selection algorithms. The

algorithm above loops 10 times (from 1 to 10), and each time it prints the square of the current integer. Our algorithm is much smaller than our first attempt, and it can be changed to print the first 1,000 squares by simply changing the 10 to 1,000. You should already be starting to see the value of iteration in our problem-solving efforts. OK, let's try a harder example:

Example 9.2. Reading a Valid GPA

Problem Description: Get a GPA from the user until the GPA is valid (0.0-4.0)

Algorithm: This algorithm is harder than the last one because this time we don't know how many times we need to loop. If the user enters a valid GPA the first time, we won't have to loop at all. On the other hand, the user may enter a number of invalid GPAs before finally entering a valid one. Let's give it a try:

```
-- Prompt for and get the GPA
-- While the GPA is less than 0.0 or greater than 4.0
  -- Print an error message
  -- Re-prompt for and get the GPA
```

The first thing we do in the algorithm is prompt for and get the first GPA from the user. If the GPA is valid (between 0.0 and 4.0 inclusive), we skip the loop body because the GPA is not less than 0.0 nor is it greater than 4.0. If the user enters an invalid GPA, however, we enter the loop body, print an error message, and prompt for and get a new GPA from the user. We'll keep looping as long as the user enters invalid GPAs - when they finally enter a valid GPA, the loop stops.

We've looked at a couple of problems in which iteration helped us devise a fairly elegant solution. Now let's see how we can implement these solutions in Ada.

9.1. For Loops

In our first example, printing the squares of the integers from 1 to 10, we knew how many times the loop should execute. This kind of loop is called a *count-controlled loop*, since we can figure out how many times to loop

by simply using a counter. To implement algorithms solving problems in which we know how many times a loop should execute before we get to it, we use a *for loop*. The *for* loop syntax is described below.

For Loop Syntax

```
for <loop control variable> in <lower bound> ..
                                <upper bound> loop

    <loop body>

end loop;
```

<loop control variable> - this is the name of the "variable that controls the loop". The loop control variable is incremented by one each time through the loop.³

<lower bound> - the initial value given to the loop control variable.

<upper bound> - the final value of the loop control variable. The loop body executes one more time when the loop control variable = upper bound, then the loop terminates.

<loop body> - the code that's executed each time through the loop.

A couple of notes about the *for* loop are in order before we try one out. First of all, there's only one kind of variable that you can use in an Ada program without declaring it - the loop control variable in a *for* loop. You should NEVER declare the loop control variable for the *for* loop as a variable in your program, because the compiler treats it as a different variable anyway! These are the only variables you get "for free", so you might as well take advantage of them.

Second, because you never declare the loop control variable for a *for* loop, the compiler has to figure out what data type the loop control variable is. It does this by looking at the data types of the lower bound and upper bound of the loop. Ada will actually let us use a variety of data

³Strictly speaking, the loop control variable is actually called the loop parameter in Ada 95. To avoid confusion with procedure parameters, we choose to use the term loop control variable instead.

types for these bounds, but for the problems in this book, integers will do fine as the bounds on our `for` loops.

Finally, in the `for` loop described above, the loop control variable "counts" up from the lower bound to the upper bound. In some cases, we might actually want to count down instead, making the loop control variable start at the upper bound, decrement by one each time through the loop, and end at the lower bound. We can accomplish that by adding a `reverse` to the `for` loop as follows:

```
for <loop control variable> in reverse <lower bound> ..
                                <upper bound> loop
```

All right, ready to try one? Let's start by copying our algorithm from Example 9.1:

```
-- For each of the integers from 1 to 10
-- Print the square of that integer
```

First we'll add the start and end of the `for` loop:

```
-- For each of the integers from 1 to 10
for Square_Integer in 1 .. 10 loop

    -- Print the square of that integer

end loop;
```

`Square_Integer` is our loop control variable, and it will start at 1 and go up to 10. Now we add code for the body of the loop, yielding:

```
-- For each of the integers from 1 to 10
for Square_Integer in 1 .. 10 loop

    -- Print the square of that integer
    Put (Item => Square_Integer ** 2,
        Width => 3);
    New_Line;

end loop;
```

The output when we run this code fragment will look like:

```
1
4
9
16
25
36
49
64
81
100
```

The loop executes 10 times, just as we wanted, printing the square of the integers from 1 to 10. Pretty straightforward, right?

Now, you may have wondered about our wording when we said "To implement algorithms solving problems in which we know how many times a loop should execute before we get to it, we use the *Ada for loop*." Specifically, why did we say "before we get to it"? It turns out that we don't have to know how many times the *for* loop will execute when we write the program - we just have to make sure we know how many times the *for* loop will execute before the program reaches that point in its execution. Consider the following example:

Example 9.3. Printing Squares of the Integers from 1 to n

Problem Description: Write a program that will print the squares of the integers from 1 to *n*, where *n* is provided by the user.

Algorithm: Our algorithm looks a lot like the one from Example 9.1 - only minor modifications are required:

```
-- Prompt for and get n
-- For each of the integers from 1 to n
  -- Print the square of that integer
```

And when we implement the algorithm in Ada, we get:

```

-- Prompt for and get n
Put (Item => "Please enter a value for N ");
Put (Item => "(1 or greater) : ");
Get (Item => N);
Skip_Line;

-- For each of the integers from 1 to n
for Square_Integer in 1 .. N loop

    -- Print the square of that integer
    Put (Item => Square_Integer ** 2,
        Width => 3);
    New_Line;

end loop;

```

So that's it - when we know how many times to loop, either when we write the program or simply before we get to the loop during program execution, we use a *for* loop. There are some cases, however, when we don't know how many times to loop; the next section describes an Ada construct that helps us in those situations.

9.2. While Loops

When we know our program may have to loop, but we don't know how many times, we use a *while* loop. This kind of loop is called a *condition-controlled loop*, because we loop based on some condition rather than a counter. The syntax for *while* loops is described on the following page.

Let's implement our algorithm from Example 9.2. using a *while* loop. Our algorithm is:

```

-- Prompt for and get the GPA
-- While the GPA is less than 0.0 or greater than 4.0
    -- Print an error message
    -- Re-prompt for and get the GPA

```

Let's add the *while* loop portion (since that's our new concept), then we'll fill in the rest.

```
-- Prompt for and get the GPA

-- While the GPA is less than 0.0 or greater than 4.0
while ( GPA < 0.0 ) or ( GPA > 4.0 ) loop

    -- Print an error message
    -- Re-prompt for and get the GPA

end loop;
```

While Loop Syntax

```
while <Boolean expression> loop

    <body of the loop>

end loop;
```

<Boolean expression> - this is just like the Boolean expressions we used in our if statements; it simply evaluates to True or False. The Boolean expression is evaluated each time we get to the `while` part of the loop. If it's True, we go into the loop and execute the loop body; if it's False, we exit the loop. The Boolean expression for a while loop should contain at least one variable.

<loop body> - the code that's executed each time through the loop.

Can you see that the value of GPA determines whether we keep looping or exit the loop? OK, let's implement the rest of the algorithm:

```
-- Prompt for and get the GPA
Put (Item => "Please enter a GPA (0.0-4.0) : ");
Get (Item => GPA);
Skip_Line;

-- While the GPA is less than 0.0 or greater than 4.0
while ( GPA < 0.0 ) or ( GPA > 4.0 ) loop

    -- Print an error message
    Put (Item => "GPA MUST be between 0.0 and 4.0 !!!");
    New_Line;
```

```

-- Re-prompt for and get the GPA
Put (Item => "Please enter a GPA (0.0-4.0) : ");
Get (Item => GPA);
Skip_Line;

end loop;

```

The first thing we do is get the GPA from the user, then while the GPA is invalid we print an error message and get a new GPA from the user. How many times does the body of the while loop execute? It depends on the user. If the user enters a valid GPA the first time, the body of the loop is never executed. If the user enters an invalid GPA first, then enters a valid GPA, the body of the loop executes once. You get the idea.

There are three things we need to do with the variables in the Boolean expression for every while loop we write, and we can remember those things with the acronym **I.T.M.** Before we get to the loop, we need to **Initialize** the variables contained in the Boolean expression so that they have values before we get to the line starting with while, which **Tests** the Boolean expression to see if the loop body should execute or not. Finally, within the loop body, we need to **Modify** at least one of the variables in the Boolean expression (give it a new value). So for every while loop, we need to make sure the variables in the Boolean expression are Initialized, Tested, and Modified - I.T.M.

So what happens if we forget about I.T.M.? If we don't initialize the variables in the Boolean expression before the loop, the test in the while part is simply based on whatever happens to be in those memory locations (and the chances of those being exactly the values we want are pretty slim). If we don't properly test the Boolean expression, the decision about whether to loop or not won't be based on the condition we really want. And if we don't modify at least one of the variables in the Boolean expression inside the loop body, the loop will go on forever (commonly called an *infinite loop*)! Here's why - if we get into the loop, the Boolean expression in the while part must have been True. If we don't change at least one of the variables inside the loop body, the Boolean expression is still True, so we loop again, and again, and again ...

You might be wondering why we didn't have to worry about I.T.M. in our `for` loops. The reason is simple - Ada handles it for us. The loop control variable for a `for` loop is automatically initialized to the lower bound, automatically tested each time through the loop to see if it's reached the upper bound, and automatically modified each time through the loop (specifically, incremented by 1).

The `while` loop gives us the ability to implement a condition-controlled loop, so we can implement algorithms in which we don't know how many times we need to loop. There is an alternative to the `while` loop, though -- we discuss this alternative below.

9.3. Basic Loops

Although a `while` loop can be used to implement any condition-controlled loop, some people prefer using a more general loop structure, called simply a *basic loop*. The syntax for basic loops is described below.

Basic Loop Syntax

```
loop
    <first part of loop body>

    exit when <Boolean expression>;

    <second part of loop body>
end loop;
```

<Boolean expression> - The Boolean expression is evaluated each time we get to the `exit when` part of the loop. If it's True, we exit from the loop; if it's False, we continue to the next statement in the loop body. The Boolean expression should contain at least one variable.

<first part of loop body> - the code that's executed before the `exit when` each time through the loop.

<second part of loop body> - the code that's executed after the `exit when` each time through the loop.

Note that, if we put the `exit` statement at the beginning of the loop body, the loop is equivalent to a while loop. We can also put the `exit` statement in the middle of the loop body, at the end of the loop body, or even in multiple places in the loop body (though this last option can get confusing). Let's rewrite our algorithm from Example 9.2:

```
-- Loop
-- Prompt for and get the GPA
-- Exit when the GPA is >= 0.0 and <= 4.0
-- Print an error message
```

When we implement this algorithm in Ada, we get:

```
-- Loop
loop

    -- Prompt for and get the GPA
    Put (Item => "Please enter a GPA (0.0-4.0) : ");
    Get (Item => GPA);
    Skip_Line;

    -- Exit when the GPA is >= 0.0 and <= 4.0
    exit when ( GPA >= 0.0 ) and ( GPA <= 4.0 );

    -- Print an error message
    Put (Item => "GPA MUST be between 0.0 and 4.0 !!!");
    New_Line;

end loop;
```

The loop body (at least the first part of it) always executes at least once. The code prompts for and gets a GPA, then repeatedly prints an error message and gets a new GPA until the GPA is valid, at which point the loop is exited.

9.4. Putting It All Together

Now let's go through the entire problem-solving process for a problem that needs to use iteration. Here's the problem description:

Read in a set of 10 test scores and calculate and print the average. Each score must be between 0 and 100.

Understand the Problem

Do we understand the problem? It seems pretty straightforward, so let's move on to the next step.

Design a Solution

Remember, this consists of two steps. The first step is to use top-down decomposition to break the problem into smaller subproblems. The second step is to write detailed algorithms to solve each of the subproblems.

To help us decide what the subproblems should be, let's write a high-level algorithm for our solution:

```
-- Print introduction
-- Set Sum to 0
-- Loop 10 times
    -- Prompt for and get Score
    -- Add Score to Sum
-- Calculate Average as Sum/10
-- Print out average
```

Some of these steps seem small enough already (like setting Sum to 0 and adding Score to Sum) while others look like good candidates for subproblems. Specifically, let's break the problem into the following subproblems:

```
Print Introduction
Get Score
Print Average
```

Since we know that we'll implement each of these subproblems as a procedure, let's write the comment blocks (including the detailed algorithms) for each of the procedures:

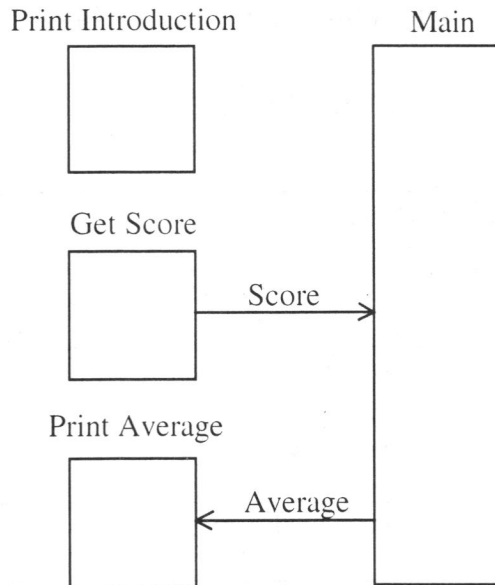
```
-----
--
-- Name : Print_Introduction
-- Description : This procedure prints the introduction
--               for the program
-- Algorithm :
--               Print the introduction
--
-----

-----
--
-- Name : Get_Score
-- Description : This procedure gets a score from the
--               user, ensuring that the score is between 0 and 100
-- Algorithm :
--               Prompt for and get score
--               While the score is less than 0 or greater than 100
--                   Print an error message
--                   Re-prompt for and get score
--
-----

-----
--
-- Name : Print_Average
-- Description : This procedure prints
-- Algorithm :
--               Print the average score
--
-----
```

Before we move on to the next step, there's one more thing to do in the design - decide what information needs to go in to and out of each procedure. The `Print_Introduction` procedure doesn't need any parameters. Because the `Get_Score` procedure gets a valid score from the user, it should pass that score out to the main program before terminating.

`Print_Average` needs the average to print it, so the average should get passed in to this procedure from the main program. The resulting decomposition diagram is as follows:



Let's write the procedure headers for each of these procedures so we remember what we've decided for the information flow:

```
procedure Print_Introduction is
```

```
procedure Get_Score ( Score : out Integer ) is
```

```
procedure Print_Average ( Average : in Float ) is
```

Now we're ready to move on to our next step.

Write the Test Plan

Recall that to test code that contains a `while` loop, we want to make sure we execute the loop body 0 times, once, and multiple times in our testing. The boundary values we need to include in our testing for this program are -1, 0, 1, 99, 100, and 101. In the test case below, we execute the loop body 0 times for the first score, once for the second score, multiple times

(twice) for the third score, and 0 times for the remaining scores. You can easily see where we've included the boundary values in the test case.

**Test Case 1 : Executing Loop body 0, 1, and Multiple Times,
Checking All Boundary Values**

Step	Input	Expected Results	Actual Results
1	0 for first score	Prompt for second score	
2	-1 for second score	Error message, reprompt	
3	1 for second score	Prompt for third score	
4	101 for third score	Error message, reprompt	
5	-1 for third score	Error message, reprompt	
6	100 for third score	Prompt for fourth score	
7	99 for fourth score	Prompt for fifth score	
8	70 for fifth score	Prompt for sixth score	
9	75 for sixth score	Prompt for seventh score	
10	80 for seventh score	Prompt for eighth score	
11	85 for eighth score	Prompt for ninth score	
12	90 for ninth score	Prompt for tenth score	
13	95 for tenth score	Print 69.50 for average	

Write the Code

Because we did such a careful job with our top-down decomposition and algorithm development, writing the code is easy. Here's what it looks like:

```
-----
--
-- Author : Joe Strummer
-- Description : This program reads in a set of 10 test
--               scores and calculates and prints the average. Each
--               score must be between 0 and 100.
-- Algorithm :
--   Print introduction
--   Set Sum to 0
--   Loop 10 times
--     Prompt for and get Score
--     Add Score to Sum
--   Calculate Average as Sum/10
--   Print out average
--
-----
```

```
with Ada.Text_IO;
use  Ada.Text_IO;
```

```
with Ada.Integer_Text_IO;
use  Ada.Integer_Text_IO;
```

```
with Ada.Float_Text_IO;
use  Ada.Float_Text_IO;
```

```
procedure Average_Scores is
```

```
-----
--
-- Name : Print_Introduction
-- Description : This procedure prints the introduction
--               for the program
-- Algorithm :
--   Print the introduction
--
-----
```

```

procedure Print_Introduction is
begin
    --      Print the introduction
    Put (Item => "This program reads in a set of 10 ");
    Put (Item => "test scores and calculates and");
    New_Line;
    Put (Item => "prints the average.  Each score must ");
    Put (Item => "be between 0 and 100.");
    New_Line;
    New_Line;

end Print_Introduction;

-----
--
-- Name : Get_Score
-- Description : This procedure gets a score from the
--      user, ensuring that the score is between 0 and 100
-- Algorithm :
--      Prompt for and get score
--      While the score is less than 0 or greater than 100
--          Print an error message
--          Re-prompt for and get score
--
-----

procedure Get_Score ( Score : out Integer ) is
begin
    --      Prompt for and get score
    Put (Item => "Please enter score (0-100) : ");
    Get (Item => Score );
    Skip_Line;

    --      While the score is less than 0 or
    --      greater than 100
    while ( Score < 0 ) or ( Score > 100 ) loop

        --          Print an error message
        Put (Item => "Score must be between 0 and 100!!!");
        New_Line;
    end loop;
end Get_Score;

```

```

--      Re-prompt for and get score
Put (Item => "Please enter score (0-100) : ");
Get (Item => Score );
Skip_Line;

end loop;

end Get_Score;

-----
--
-- Name : Print_Average
-- Description : This procedure prints
-- Algorithm :
--   Print the average score
--
-----

procedure Print_Average ( Average : in Float ) is
begin

--   Print the average score
Put (Item => "Average test score is : ");
Put (Item => Average,
    Fore => 3,
    Aft  => 2,
    Exp  => 0);
New_Line;

end Print_Average;

-----
-- START OF MAIN PROGRAM
-----

Sum           : Integer;
Current_Score : Integer;
Average       : Float;

begin

--   Print introduction
Print_Introduction;

```



```

--      Set Sum to 0
Sum := 0;

--      Loop 10 times
for Counter in 1 .. 10 loop

    --      Prompt for and get Score
    Get_Score (Score => Current_Score);

    --      Add Score to Sum
    Sum := Sum + Current_Score;

end loop;

--      Calculate Average as Sum/10
Average := Float (Sum) / 10.0;

--      Print out average
Print_Average (Average => Average);

end Average_Scores;

```

The only thing that causes us some trouble in our code is our choice to have the `Sum` stored as an integer. Although this makes intuitive sense (since all the test scores are integers), it causes a type compatibility problem when we try to calculate the `Average`. To solve this problem, we use something called a *type cast* in the line

```
Average := Float (Sum) / 10.0;
```

Essentially, writing `Float (Sum)` means "convert `Sum` to a float just for this computation". Of course, another way to work around this problem would be to simply force the scores and the sum to be floats, but this solution is less desirable because scores are actually integers and we should store and use them as such.

Test the Code

So now we need to test our code to make sure the program works. When we run the program above and fill in the actual results in our test plan, we get the following (which is exactly what we expected):

**Test Case 1 : Executing Loop Body 0, 1, and Multiple Times,
Checking All Boundary Values**

Step	Input	Expected Results	Actual Results
1	0 for first score	Prompt for second score	Prompts for second score
2	-1 for second score	Error message, reprompt	Error message, reprompts
3	1 for second score	Prompt for third score	Prompts for third score
4	101 for third score	Error message, reprompt	Error message, reprompts
5	-1 for third score	Error message, reprompt	Error message, reprompts
6	100 for third score	Prompt for fourth score	Prompts for fourth score
7	99 for fourth score	Prompt for fifth score	Prompts for fifth score
8	70 for fifth score	Prompt for sixth score	Prompts for sixth score
9	75 for sixth score	Prompt for seventh score	Prompts for seventh score
10	80 for seventh score	Prompt for eighth score	Prompts for eighth score
11	85 for eighth score	Prompt for ninth score	Prompts for ninth score
12	90 for ninth score	Prompt for tenth score	Prompts for tenth score
13	95 for tenth score	Print 69.50 for average	Prints 69.50 for average

9.5. Common Mistakes

Declaring For Loop Control Variable

Remember we said that there's one and only one time you can use a variable in Ada without explicitly declaring it? That variable is the loop control variable in a `for` loop. If you DO try to declare the loop control variable as a variable, the compiler treats that as a different variable from the loop control variable, even though they have the same name! You should therefore never declare the loop control variable in a `for` loop as a variable.

Forgetting I.T.M. in a While Loop

The most common cause of infinite loops (loops that just keep going) is the programmer forgetting about I.T.M. You have to make sure all the variables contained in the Boolean expression for the loop are initialized so that they have values before we get to the `while` part, which tests the Boolean expression to see if the loop body should execute or not. We also need to make sure that at least one of the variables in the Boolean expression is modified within the loop body. If (when) you write a program containing an infinite loop, check I.T.M.

Using an If Statement Instead of a While Loop

Some people have a tendency to use an `if` statement when they're trying to get valid user inputs. This would only work if we had a GUARANTEE that the user would never enter two invalid inputs in a row. We'd need that guarantee because there's no way for an `if` statement to "loop back", so we could never catch the second invalid input. A `while` loop is definitely the way to go when we're trying to get valid user inputs.

9.6. Exercises

1. Write a program that prints out the first ten integers and the squares of the first ten integers in a table. For example:

Number	Square
1	1
2	4
...	...

2. Write a program that prints the first 20 numbers in the Fibonacci sequence. This sequence starts as:

1 1 2 3 5 8 ...

and you get each number in the sequence as the sum of the previous two numbers in the sequence.

3. Write a program that reads in 3 test percentages and prints the average. Test percentages must be between 0.0 and 100.0.

4. Write a program that repeatedly asks the user if they want to continue, then terminates when they answer n or N.

5. Write a program that prints numbers in the Fibonacci sequence until it reaches (and prints) 55.

Chapter 10. Arrays *- need loops to implement Arrays*

We now know a lot of Ada constructs - we know how to use procedures to help us decompose the problem, we know how to perform selection and iteration, and we know how to do more basic things, like declare variables and constants and perform input and output with a number of data types. Strictly speaking, that's sufficient to solve most of the problems we'd encounter in an introductory course, but there may be some cases in which simply using the above knowledge would be pretty awkward.

For example, suppose we had 12,000 students at our university, and we wanted to store the GPAs for all of them. We already know how to read them in (a `for` loop should have leapt to your mind), but where do we put them all? With our current knowledge we could declare 12,000 distinct variables, one for each GPA, but there's got to be a better way! Luckily, there is -- we can use an *array* to store all the GPAs, and we only need to do a little more work than we do when we declare any other kind of variable. Let's take a closer look.

10.1. Defining Array Types and Declaring Array Variables

The reason we have to do a little more work is because we need to declare our very own data type! We do this so we can declare variables (and formal parameters) of our new type. The syntax box on the following page shows how we go about doing this.

To make this a little more concrete, let's work through an example.

Example 10.1. Storing GPAs for 12,000 students

Problem Description: Set up the data type and variable required to store 12,000 GPAs.

Our first step is to define the constant for the array size. Using a constant for this will make it easier to change the array size if necessary (say, if enrollment skyrockets to 12,001) because we only have to change the array size in one place - in this constant. Here's one way to do this:

Array Type Definition and Variable Declaration

```
<constant for array size> : constant Integer :=
                                <array size>;
type <type name> is array (1 .. <constant for array size>)
                                of <element type>;
```

```
<variable name> : <type name>;
```

<constant for array size> - this is the name of the constant that stores the size of the array.

<array size> - the actual size of the array.

<type name> - the name of the new type we're defining.

<element type> - the data type for each element of the array.

<variable name> - the name of a variable declared as our array type.

```
Num_Students : constant Integer := 12000;
```

The next thing we need to do is define the new data type:

```
type GPA_Array is array (1..Num_Students) of Float;
```

With the above type definition, our array will consist of 12,000 "boxes" or *elements*. We've decided that the elements of the array should be numbered from 1 to Num_Students -- the number for a particular element is called the *index* of that element. Although using integers to index the elements of the array was a clear choice for this problem, we can also index elements of arrays using characters (or even Booleans) instead. Each element will be a Float, which also seemed to be a clear choice given that each element holds a GPA. No matter what data type we pick for the array elements, every single element of the array will be of that type. In other words, we can't have some Float elements, some Integer elements, etc. in a single array. OK, our last step is to declare a variable of our new type:

```
GPAs : GPA_Array;
```

Now we actually have memory allocated for a variable called `GPAs`; `GPAs` is an array of 12,000 floating point numbers. Note that memory is NOT allocated when we define the array type; rather memory is allocated each time we declare a variable of that type.

So that's how we set up our data type and variable for an array. Our next step is to show how we can actually get at elements of the array to do useful things like input, output, and calculations with those elements.

10.2. Accessing Elements of the Array

Now that we have a variable for the array, how do we use it? Well, we use the variable in the same way we've used other variables - the only real difference is that we do assignments, input, output, and calculations with single elements of the array rather than the entire array. Let's make this a little more concrete; the syntax box on the following page shows us the proper syntax for these operations.

The only difference between array variables and more "normal" variables is that we have to say which element of the array we want to use. For example, if we wanted to set the first element of our `GPAs` array to 0.0, we could simply say:

```
GPAs(1) := 0.0;
```

Similarly, if we wanted to read into the 151st element of the `GPAs` array, we could use:

```
Get (Item => GPAs(151));
```

And so on for output and calculations.

Array Element Operations*Assignment*

```
<variable name>(<index>) := <value>;
```

<variable name> - the name of a variable declared as our array type.

<index> - the number of the element in which we want to store <value>.

<value> - the value we want to put into that element.

Input

```
Get (Item => <variable name>(<index>));
```

<variable name> - the name of a variable declared as our array type.

<index> - the number of the element in which we want to store the user input.

Output

```
Put (Item => <variable name>(<index>));
```

<variable name> - the name of a variable declared as our array type.

<index> - the number of the element we want to output to the screen.

Calculations

```
Sum := Sum + <variable name>(<index>);
```

<variable name> - the name of a variable declared as our array type.

<index> - the number of the element we want to add to Sum.

10.3. Arrays and For Loops - "Walking the Array"

Although there are certainly times when we want to use specific elements of an array, there are many other times that we want to look at all the elements in the array -- in some sense, "walking the array" from top to bottom. For example, let's build on Example 10.1. with the following example:

Example 10.2. Averaging the GPAs of 12,000 Students

Problem Description: Calculate the average of the 12,000 GPAs in the array in Example 10.1 (assume the array has already been filled with GPAs).

First, let's come up with an algorithm:

```
-- Initialize Sum to 0.0
-- Loop from 1 to number of students
--   Set Sum to Sum + current GPA
-- Set Average to Sum / number of students
```

Then we convert this algorithm into code:

```
-- Initialize Sum to 0.0
Sum := 0.0;

-- Loop from 1 to number of students
for Index in 1 .. Num_Students loop

    --   Set Sum to Sum + current GPA
    Sum := Sum + GPAs(Index);

end loop;

-- Set Average to Sum / number of students
Average := Sum / Float (Num_Students);
```

A couple of things are worth noting. First of all, we used a `for` loop to walk the array. That's because we know by the time we get to the loop exactly how big the array is, so we know how many times to loop to look at each element once. Arrays and `for` loops go together very well, and you'll find that you almost always use a `for` loop to walk an array.

Secondly, we used `Num_Students` as the upper bound of the `for` loop and the divisor in our calculation of the average. As we mentioned earlier, this makes it easier to change our program if the number of students changes. For instance, simply changing the `Num_Students` constant at the top of our program would change the size of the array, the upper bound of

the `for` loop, and the divisor in the calculation of the average. You may have also noticed that we used a type cast in the calculation of the average. `Num_Students` should clearly be an `Integer` (we can only have a "whole number" of students), and the `Sum` of the GPAs just as clearly needs to be a `Float` (because each GPA is a `Float`), so using a type cast is the cleanest way to perform the calculation.

Finally, when we reference elements of the `GPAs` array, we use `GPAs(Index)`. In our examples in the previous section, we used numeric literals for our indices, but it's certainly just as valid to use a variable to index into the array. The first time through the `for` loop, `Index` is 1 so we add the first element of the `GPAs` array to `Sum`, the second time through the `for` loop, `Index` is 2 so we add the second element of the `GPAs` array to `Sum`, and so on. You can see why `for` loops are so useful for walking arrays, since by choosing appropriate lower and upper bounds for the loop we can ensure that we use each element of the array once.

10.4. Multi-Dimensional Arrays

So far, the only arrays we've dealt with have been one-dimensional (a single column of elements). Can we make multi-dimensional arrays in Ada? Sure we can! Let's say we wanted an array with 4 rows and 5 columns of `Integers`. Here's how we'd define our constants and the array type:

```
Num_Rows : constant Integer := 4;
Num_Columns : constant Integer := 5;
type Two_D_Array_Type is
    array (1..Num_Rows, 1..Num_Columns) of Integer;
```

Note that we now specify the range of values for two dimensions of indices (rows and columns). If we have a variable declared as:

```
Two_D_Array : Two_D_Array_Type;
```

then we access elements of this array using the general form:

```
Two_D_Array(<row number>, <column number>)
```

For example, to put a 0 into the upper left corner of the array (row 1, column 1), we'd say:

```
Two_D_Array(1,1) := 0;
```

Columns are numbered from left to right, and rows are numbered from top to bottom. Let's look at an example.

Example 10.3. Initializing Elements to 0

Problem Description: Initialize all the elements of `Two_D_Array` to 0.

First, let's come up with an algorithm:

```
-- Loop from 1 to Num_Rows
-- Loop from 1 to Num_Columns
-- Set current element of Two_D_Array to 0
```

We're still using `for` loops to walk the array, but this time we're using nested `for` loops because we need to walk both the rows and the columns. Here's the code for our algorithm:

```
-- Loop from 1 to Num_Rows
for Row in 1 .. Num_Rows loop

    -- Loop from 1 to Num_Columns
    for Column in 1 .. Num_Columns loop

        -- Set current element of Two_D_Array to 0
        Two_D_Array(Row,Column) := 0;

    end loop;

end loop;
```

And that's all there is to using two-dimensional arrays. We're not limited to two-dimensional arrays, either; we can essentially use as many array dimensions as we want!

10.5. Putting It All Together

Let's go through the entire problem-solving process for a problem in which arrays are the most reasonable way to store the data. Here's the problem description:

Read in a set of 10 test scores and calculate and print the average. Also determine what the highest and lowest scores in the set are. Each score must be between 0 and 100.

Understand the Problem

Well, the problem seems to be well-specified. There's no explicit requirement to use arrays, and we solved a similar problem in the previous chapter without using arrays, but we'll find that using an array to store all the scores will make our problem decomposition easier.

Design a Solution

To help us decide what the subproblems should be, let's write a high-level algorithm for our solution:

```
-- Print introduction
-- Read in scores
-- Calculate average
-- Find high score
-- Find low score
-- Print out average, high score, and low score
```

Based on our high-level algorithm, let's break the problem into the following subproblems:

- Print Introduction
- Get Scores
- Calculate Average
- Find High Score
- Find Low Score
- Print Average, High Score, and Low Score

Since we know that we'll implement each of these subproblems as a procedure, let's write the comment blocks (including the detailed algorithms) for each of the procedures:

```
-----
--
-- Name : Print_Introduction
-- Description : This procedure prints the introduction
--               for the program
-- Algorithm :
--               Print the introduction
--
-----
```

```
-----
--
-- Name : Get_Scores
-- Description : This procedure gets the scores from the
--               user, ensuring that each score is between 0 and 100
-- Algorithm :
--               Loop from 1 to number of scores
--                   Prompt for and get score
--                   While the score is less than 0 or
--                       greater than 100
--                       Print an error message
--                       Re-prompt for and get score
--
-----
```

```
-----
--
-- Name : Calculate_Average
-- Description : This procedure calculates the average of
--               the test scores
-- Algorithm :
--               Initialize Sum to 0
--               Loop from 1 to number of scores
--                   Set Sum to Sum + current score
--               Set Average to Sum / number of scores
--
-----
```

```

-----
--
-- Name : Find_High_Score
-- Description : This procedure finds the highest test
--               score
-- Algorithm :
--   Initialize High_Score to first score in array
--   Loop from 2 to number of scores
--       If current score is greater than High_Score
--           Set High_Score to current score
--
-----

-----
--
-- Name : Find_Low_Score
-- Description : This procedure finds the lowest test
--               score
-- Algorithm :
--   Initialize Low_Score to first score in array
--   Loop from 2 to number of scores
--       If current score is less than Low_Score
--           Set Low_Score to current score
--
-----

-----
--
-- Name : Print_Average_High_Low
-- Description : This procedure prints the average score,
--               the high score, and the low score
-- Algorithm :
--   Print the average score
--   Print the high score
--   Print the low score
--
-----

```

We've seen most of this before, but there's certainly something new in the `Find_High_Score` and `Find_Low_Score` procedures (which are remarkably similar to each other). Here's the idea behind the algorithm for `Find_High_Score` -- we start by assuming that the first score is the highest

score. We then walk the array (starting at the second element), and we check each element to see if it's higher than the current high score. If it isn't we don't do anything, but if it is we replace the high score with the current score. `Find_Low_Score` works in a similar manner. You may want to walk through the algorithms a few times with some small sample inputs (say once with an array in which the high score is in the first element of the array, once with an array in which the high score is in one of the middle elements, and once with an array in which the high score is in the last element of the array).

Now let's decide what information needs to go in to and out of each procedure. The `Print_Introduction` procedure doesn't need any parameters. Because the `Get_Scores` procedure gets 10 valid scores from the user, it should pass the array of those scores out to the main program before terminating. `Calculate_Average` needs the array of scores to come in to the procedure, and once the average has been calculated the procedure should pass the average out to the main program. `Find_High_Score` needs the array of scores to come in to the procedure (so it can walk the array looking for the high score), then it should pass the high score back to the main program. `Find_Low_Score` also needs the array of scores to come in to the procedure (so it can walk the array looking for the low score), then it should pass out the low score. `Print_Average_High_Low` needs the average, high score, and low score to print them, so these values should get passed in to this procedure from the main program. The resulting decomposition diagram is on the following page.

Let's write the procedure headers for each of these procedures so we remember what we've decided for the information flow:

```

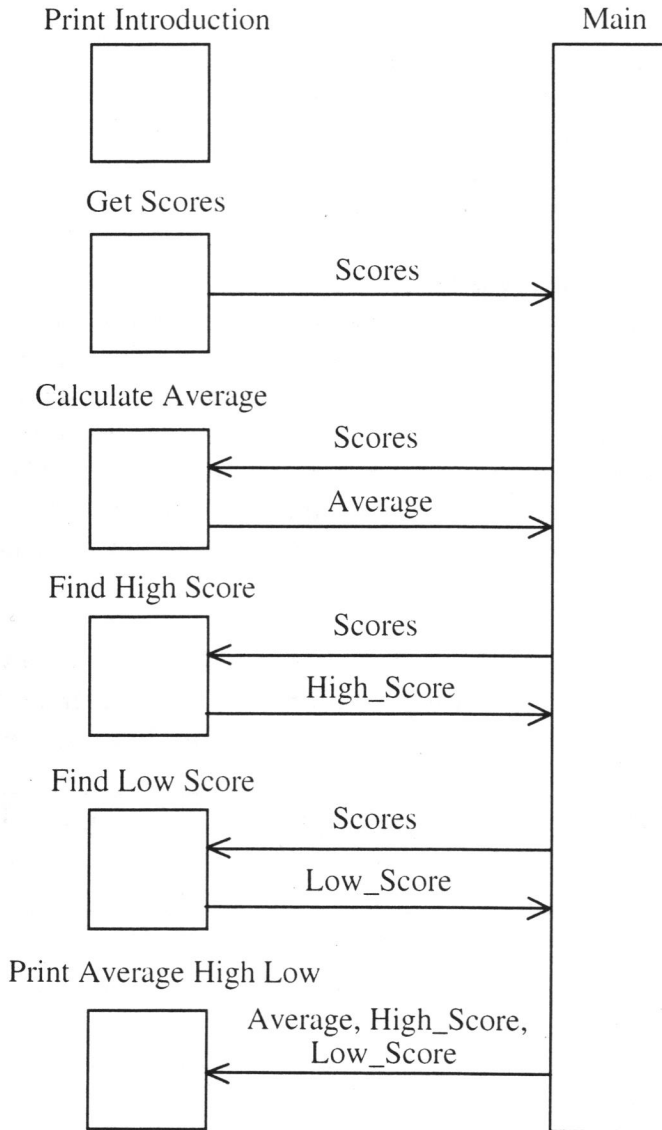
procedure Print_Introduction is

procedure Get_Scores ( Scores : out Score_Array ) is

procedure Calculate_Average ( Scores : in Score_Array;
    Average : out Float ) is

procedure Find_High_Score ( Scores : in Score_Array;
    High_Score : out Integer ) is

```



```

procedure Find_Low_Score ( Scores : in Score_Array;
    Low_Score : out Integer ) is

```

```

procedure Print_Average_High_Low ( Average : in Float;
    High_Score : in Integer;
    Low_Score : in Integer ) is

```


Now we're ready to move on to our next step.

Write the Test Plan

To thoroughly test the loop, we can simply use Test Case 1 from the problem at the end of the last chapter. Because we're using arrays, and most errors occur near the "edges" (or bounds) of an array, we should include test cases where the highest score is the first score in the array, the highest score is the last score in the array, the highest score is near the middle of the array, the lowest score is the first score in the array, the lowest score is the last score in the array, and the lowest score is near the middle of the array. Test Case 1 takes care of the lowest score as the first one and the highest score "somewhere in the middle"; Test Cases 2 and 3 take care of the other possibilities. Note that, because we've thoroughly tested the loop in Test Case 1, we don't enter any invalid inputs in Test Cases 2 and 3.

Test Case 1 : Executing Loop Body 0, 1, and Multiple Times, Checking All Boundary Values, Highest Score Near Middle, Lowest Score in First Element

Step	Input	Expected Results	Actual Results
1	0 for first score	Prompt for second score	
2	-1 for second score	Error message, reprompt	
3	1 for second score	Prompt for third score	
4	101 for third score	Error message, reprompt	
5	-1 for third score	Error message, reprompt	
6	100 for third score	Prompt for fourth score	
7	99 for fourth score	Prompt for fifth score	

8	70 for fifth score	Prompt for sixth score	
9	75 for sixth score	Prompt for seventh score	
10	80 for seventh score	Prompt for eighth score	
11	85 for eighth score	Prompt for ninth score	
12	90 for ninth score	Prompt for tenth score	
13	95 for tenth score	Print 69.50 for average, 100 for high score, 0 for low score	

Test Case 2 : Highest Score in First Element, Lowest Score in Last Element

Step	Input	Expected Results	Actual Results
1	100 for first score	Prompt for second score	
2	1 for second score	Prompt for third score	
3	95 for third score	Prompt for fourth score	
4	99 for fourth score	Prompt for fifth score	
5	70 for fifth score	Prompt for sixth score	
6	75 for sixth score	Prompt for seventh score	
7	80 for seventh score	Prompt for eighth score	
8	85 for eighth score	Prompt for ninth score	

9	90 for ninth score	Prompt for tenth score	
10	0 for tenth score	Print 69.50 for average, 100 for high score, 0 for low score	

Test Case 3 : Highest Score in Last Element, Lowest Score Near Middle

Step	Input	Expected Results	Actual Results
1	99 for first score	Prompt for second score	
2	1 for second score	Prompt for third score	
3	95 for third score	Prompt for fourth score	
4	0 for fourth score	Prompt for fifth score	
5	70 for fifth score	Prompt for sixth score	
6	75 for sixth score	Prompt for seventh score	
7	80 for seventh score	Prompt for eighth score	
8	85 for eighth score	Prompt for ninth score	
9	90 for ninth score	Prompt for tenth score	
10	100 for tenth score	Print 69.50 for average, 100 for high score, 0 for low score	

Write the Code

Here's the completed code for the program:

```

-----
--
-- Author : Joe Elliott
-- Description : This program reads in a set of 10 test
--               scores and calculates and prints the average. It also
--               determines what the highest and lowest scores in the
--               set are. Each score must be between 0 and 100.
-- Algorithm :
--   Print introduction
--   Read in scores
--   Calculate Average
--   Find high score
--   Find low score
--   Print out average, high score, and low score
--   Wait for user to press enter to end
--
-----

```

```

with Ada.Text_IO;
use  Ada.Text_IO;

```

```

with Ada.Float_Text_IO;
use  Ada.Float_Text_IO;

```

```

with Ada.Integer_Text_IO;
use  Ada.Integer_Text_IO;

```

```

procedure Process_Scores is

```

```

  Num_Scores : constant Integer := 10;
  type Score_Array is array (1..Num_Scores) of Integer;

```

```

-----
--
-- Name : Print_Introduction
-- Description : This procedure prints the introduction
--               for the program
-- Algorithm :
--   Print the introduction
--
-----

```

```

procedure Print_Introduction is
begin

```

```

    --      Print the introduction
    Put (Item => "This program reads in a set of 10 ");
    Put (Item => "test scores and calculates");
    New_Line;
    Put (Item => "and prints the average.  It also ");
    Put (Item => "determines what the highest");
    New_Line;
    Put (Item => "and lowest scores in the set are.  ");
    Put (Item => "Each score must be between");
    New_Line;
    Put (Item => "0 and 100.");
    New_Line;
    New_Line;

```

```

end Print_Introduction;

```

```

-----
--
-- Name : Get_Scores
-- Description : This procedure gets the scores from the
--      user, ensuring that each score is between 0 and 100
-- Algorithm :
--      Loop from 1 to number of scores
--      Prompt for and get score
--      While the score is less than 0 or
--          greater than 100
--          Print an error message
--          Re-prompt for and get score
--
-----

```

```

procedure Get_Scores ( Scores : out Score_Array ) is
begin

```

```

    --      Loop from 1 to number of scores
    for Index in 1 .. Num_Scores loop

```

```

--          Prompt for and get score
Put (Item => "Please enter score ");
Put (Item => Index,
    Width => 1);
Put (Item => " (0-100) : ");
Get (Item => Scores(Index));
Skip_Line;

--          While the score is less than 0 or
--          greater than 100
while ( Scores(Index) < 0 ) or
      ( Scores(Index) > 100 ) loop

    --          Print an error message
    Put (Item => "Scores must be between 0 and ");
    Put (Item => "100!!!");
    New_Line;

    --          Re-prompt for and get score
    Put (Item => "Please re-enter score ");
    Put (Item => Index,
        Width => 1);
    Put (Item => " (0-100) : ");
    Get (Item => Scores(Index));
    Skip_Line;

end loop;

end loop;

end Get_Scores;

-----
--
-- Name : Calculate_Average
-- Description : This procedure calculates the average of
-- the test scores
-- Algorithm :
--   Initialize Sum to 0
--   Loop from 1 to number of scores
--   Set Sum to Sum + current score
--   Set Average to Sum / number of scores
--

```

```
-----
procedure Calculate_Average ( Scores : in Score_Array;
    Average : out Float ) is
```

```
    Sum : Integer;
```

```
begin
```

```
    --    Initialize Sum to 0
```

```
    Sum := 0;
```

```
    --    Loop from 1 to number of scores
```

```
    for Index in 1 .. Num_Scores loop
```

```
        --    Set Sum to Sum + current score
```

```
        Sum := Sum + Scores(Index);
```

```
    end loop;
```

```
    --    Set Average to Sum / number of scores
```

```
    Average := Float(Sum) / Float(Num_Scores);
```

```
end Calculate_Average;
```

```
-----
--
```

```
-- Name : Find_High_Score
```

```
-- Description : This procedure finds the highest test
--               score
```

```
-- Algorithm :
```

```
--   Initialize High_Score to first score in array
```

```
--   Loop from 2 to number of scores
```

```
--       If current score is greater than High_Score
```

```
--           Set High_Score to current score
```

```
--
-----
```

```
procedure Find_High_Score ( Scores : in Score_Array;
```

```
    High_Score : out Integer ) is
```

```
begin
```

```

--      Initialize High_Score to first score in array
High_Score := Scores(1);

--      Loop from 2 to number of scores
for Index in 2 .. Num_Scores loop

    --          If current score is greater than
    --          High_Score
    if ( Scores(Index) > High_Score ) then

        --          Set High_Score to current score
        High_Score := Scores(Index);

    end if;

end loop;

end Find_High_Score;

```

```

-----
--
-- Name : Find_Low_Score
-- Description : This procedure finds the lowest test
-- score
-- Algorithm :
--   Initialize Low_Score to first score in array
--   Loop from 2 to number of scores
--       If current score is less than Low_Score
--       Set Low_Score to current score
--
-----

```

```

procedure Find_Low_Score ( Scores : in Score_Array;
    Low_Score : out Integer ) is
begin

    --      Initialize Low_Score to first score in array
    Low_Score := Scores(1);

    --      Loop from 2 to number of scores
    for Index in 2 .. Num_Scores loop

```



```

--      If current score is less than Low_Score
if ( Scores(Index) < Low_Score ) then

    --      Set Low_Score to current score
    Low_Score := Scores(Index);

end if;

end loop;

end Find_Low_Score;

-----
--
-- Name : Print_Average_High_Low
-- Description : This procedure prints the average score,
--      the high score, and the low score
-- Algorithm :
--      Print the average score
--      Print the high score
--      Print the low score
--
-----

procedure Print_Average_High_Low ( Average : in Float;
    High_Score : in Integer;
    Low_Score : in Integer ) is
begin

    --      Print the average score
    New_Line;
    Put (Item => "Average test score : ");
    Put (Item => Average,
        Fore => 3,
        Aft  => 2,
        Exp  => 0);
    New_Line;

    --      Print the high score
    Put (Item => "High test score      : ");
    Put (Item => High_Score,
        Width => 3);
    New_Line;

```

```

--      Print the low score
Put (Item => "Low test score      : ");
Put (Item => Low_Score,
     Width => 3);
New_Line;
New_Line;

```

```

end Print_Average_High_Low;

```

```

-----
-- START OF MAIN PROGRAM
-----

```

```

Scores      : Score_Array;
Average     : Float;
High_Score  : Integer;
Low_Score   : Integer;

```

```

begin

```

```

--      Print introduction
Print_Introduction;

```

```

--      Read in scores
Get_Scores ( Scores => Scores );

```

```

--      Calculate Average
Calculate_Average ( Scores => Scores,
                   Average => Average );

```

```

--      Find high score
Find_High_Score ( Scores => Scores,
                 High_Score => High_Score);

```

```

--      Find low score
Find_Low_Score ( Scores => Scores,
                Low_Score => Low_Score);

```

```

--      Print out average, high score, and low score
Print_Average_High_Low ( Average => Average,
                        High_Score => High_Score,
                        Low_Score  => Low_Score);

```

```
end Process_Scores;
```

Notice that, even though we continue to declare our variables just before the begin for the main program, in this program we put our constant and array type definition BEFORE the procedures. Why? Because we want the procedures to be able to use the constant as an upper bound in `for` loops, and because we want some of the parameters to be the array type. The compiler can't "know about" the constant and array type before it sees how they're defined, so they have to come before the procedures.

Test the Code

And when we run the code, our actual results match our expected results:

Test Case 1 : Executing Loop Body 0, 1, and Multiple Times, Checking All Boundary Values, Highest Score Near Middle, Lowest Score in First Element

Step	Input	Expected Results	Actual Results
1	0 for first score	Prompt for second score	Prompts for second score
2	-1 for second score	Error message, reprompt	Error message, reprompts
3	1 for second score	Prompt for third score	Prompts for third score
4	101 for third score	Error message, reprompt	Error message, reprompts
5	-1 for third score	Error message, reprompt	Error message, reprompts
6	100 for third score	Prompt for fourth score	Prompts for fourth score
7	99 for fourth score	Prompt for fifth score	Prompts for fifth score
8	70 for fifth score	Prompt for sixth score	Prompts for sixth score

9	75 for sixth score	Prompt for seventh score	Prompts for seventh score
10	80 for seventh score	Prompt for eighth score	Prompts for eighth score
11	85 for eighth score	Prompt for ninth score	Prompts for ninth score
12	90 for ninth score	Prompt for tenth score	Prompts for tenth score
13	95 for tenth score	Print 69.50 for average, 100 for high score, 0 for low score	Prints 69.50 for average, 100 for high score, 0 for low score

Test Case 2 : Highest Score in First Element, Lowest Score in Last Element

Step	Input	Expected Results	Actual Results
1	100 for first score	Prompt for second score	Prompts for second score
2	1 for second score	Prompt for third score	Prompts for third score
3	95 for third score	Prompt for fourth score	Prompts for fourth score
4	99 for fourth score	Prompt for fifth score	Prompts for fifth score
5	70 for fifth score	Prompt for sixth score	Prompts for sixth score
6	75 for sixth score	Prompt for seventh score	Prompts for seventh score
7	80 for seventh score	Prompt for eighth score	Prompts for eighth score
8	85 for eighth score	Prompt for ninth score	Prompts for ninth score

9	90 for ninth score	Prompt for tenth score	Prompts for tenth score
10	0 for tenth score	Print 69.50 for average, 100 for high score, 0 for low score	Prints 69.50 for average, 100 for high score, 0 for low score

Test Case 3 : Highest Score in Last Element, Lowest Score Near Middle

Step	Input	Expected Results	Actual Results
1	99 for first score	Prompt for second score	Prompts for second score
2	1 for second score	Prompt for third score	Prompts for third score
3	95 for third score	Prompt for fourth score	Prompts for fourth score
4	0 for fourth score	Prompt for fifth score	Prompts for fifth score
5	70 for fifth score	Prompt for sixth score	Prompts for sixth score
6	75 for sixth score	Prompt for seventh score	Prompts for seventh score
7	80 for seventh score	Prompt for eighth score	Prompts for eighth score
8	85 for eighth score	Prompt for ninth score	Prompts for ninth score
9	90 for ninth score	Prompt for tenth score	Prompts for tenth score
10	100 for tenth score	Print 69.50 for average, 100 for high score, 0 for low score	Prints 69.50 for average, 100 for high score, 0 for low score

10.6. Common Mistakes

Indexing Past the End of the Array

It should come as no surprise that, if we try to index outside the array (i.e., look at an array element that doesn't exist), our program will blow up. For example, if our array indices go from 1 to 10 and we try to look at element 11, our program will terminate abnormally (most environments will say something like "Constraint_Error"). Indexing outside the array occurs most commonly when we are actually calculating the indices of the elements we're referencing, but this can also occur if we select incorrect bounds on a `for` loop we're using to walk the array.

Trying to Get or Put the Entire Array at Once

Because we declare array variables just like any other variables, you may be tempted to try to use a `Get` statement to read into the entire array or a `Put` statement to print the entire array. You can't do this! You have to `Get` or `Put` each element of the array separately.

10.7. Exercises

1. Write a program that reads in 10 test percentages and prints the average. Test percentages must be between 0.0 and 100.0.
2. Write a program that reads in 10 GPAs and determines how many of those students are on Dean's List (GPA at least a 3.0).
3. Write a program that reads in 5 first initials, 5 last names, and 5 ages, then prints each initial, name, and age on a separate line. You should read in ALL the information before doing the output.

Chapter 11. File IO

As part of our motivation for using arrays, we talked about storing 12,000 GPAs for the students at our university. We showed how we can store those GPAs in an array, and we showed how to read into elements of an array, but we glossed over a couple of important considerations. First of all, who in their right mind is going to sit at the keyboard and type in 12,000 GPAs? Even more importantly, what happens to those GPAs after the program terminates? They disappear! So every time we want to do something with those GPAs, we have to enter all 12,000 again! There just has to be a way to store these GPAs for later use, and there is -- *files*.

The most common type of file is called a *text file*. A text file simply consists of a set of characters. Essentially, any character we can enter from the keyboard can be read from a file instead, and any character we could print to the screen can also be output to a file. Thus, files give us a great way to store large amounts of data for input to computer programs, output from computer programs, or both.

You can really think of a file as a book. Before you can read a book, you need to open it -- the same is true of files. After you're done with a book, you should close it -- again, the same is true with files. If you want to write a book rather than reading it, the book has to be created (from paper if you're writing the old-fashioned way) -- the same is true for files. We'll be introducing some new material to let us work with files, but if you think of files as books, you'll be well on your way. Let's take a closer look.

11.1. File Variables

We already know that any time we want to store some information in our program, we need to set aside memory using variables. Although the actual data we're trying to input or output is in a file, we need to set aside memory to hold the file itself⁴. We do this by declaring a file variable as described on the next page.

⁴In fact, the variable doesn't actually hold the file, it holds a pointer to the file, but conceptually it's easier to think of the variable as actually holding the file.

File Variable Declaration

```
<variable name> : File_Type;
```

<variable name> - the name of a variable declared as our file variable.

You probably noticed that we've introduced a new data type called `File_Type`. This data type is actually found in the `Ada.Text_IO` package, so you need to make sure you with and use this package in any program that uses file variables.

11.2. Opening a File for Reading

Remember when we said that files are like books? The first thing we have to do before reading a book is open it. Similarly, we have to open a file before we can read from it. Here's the syntax:

Opening Files

```
Open ( File => <variable name>,
      Mode => In_File,
      Name => "<external file name>" );
```

<variable name> - the name of the file variable.

<external file name> - the name of the external file on the disk.

Say we have a file variable called `Input_File` and the file we want to use for input is `c:\input.dat`. We would open the file using the following:

```
Open ( File => Input_File,
      Mode => In_File,
      Name => "c:\input.dat" );
```

Just as you need to know the name of the book you're going to read (so you can open the right book), you need to know the name of the file you want to read from the disk before you can open the file. We set the mode

to `In_File` because we'll be using the file for input (rather than output). After the open is completed, `Input_File` is "holding" the opened file `c:\input.dat`, and the program can start reading from the file. One word of caution -- `Open` assumes that the external file already exists on the disk, and if that's not true your program will "blow up" (giving you a `Name_Error` because it can't find a file with the name you gave it)!

11.3. Getting Input From a File

OK, the file is open for reading; now all we have to do is read from it. It's not as hard as you might think, because we can keep using the `Get` procedure we've been using all along (with one minor modification). Say we wanted to read the first GPA from the keyboard into an array called `GPAs`. We'd use the following:

```
Put ( Item => "Please enter first GPA : ");
Get ( Item => GPAs(1) );
Skip_Line;
```

Now say we wanted to read the first GPA from a file instead. We'd tell the computer to use the file (instead of the keyboard) for input by using:

```
Get ( File => Input_File,
      Item => GPAs(1) );
```

See how easy that is? We can also make the program move to the next line in an input file by using `Skip_Line` as follows:

```
Skip_Line ( File => Input_File );
```

Instead of waiting for the user to press the <enter> key on the keyboard, the program will move to the next line in the input file. Note also that it doesn't make sense to prompt the file for an input; where would you print the prompt (the file certainly isn't looking at the screen)?

11.4. Closing the File When You're Done

You should always close a book when you're done reading (or writing) it to avoid breaking the binding. Files don't have bindings, but you should still close them when you're done so that other programs can then have access to those files. Here's the required syntax:

Closing Files

```
Close ( File => <variable name> );
```

<variable name> - the name of the file variable.

As an example, here's how we'd close the `Input_File` once we were done reading from it:

```
Close ( File => Input_File );
```


It's probably about time to go through an example.

Example 11.1. Reading in 12,000 GPAs

Problem Description: Read 12,000 GPAs from a file called `c:\input.dat` into an array called `GPAs`. Assume `Input_File` has been declared as `File_Type` and the file contains exactly `Num_GPAs` GPAs (where `Num_GPAs = 12,000`).

Here's one algorithm that should work:

```
-- Open Input_File
-- Loop from 1 to number of GPAs
--   Read current GPA from Input_File
-- Close Input_File
```



And the resulting code:

```

-- Open Input_File
Open ( File => Input_File,
      Mode => In_File,
      Name => "c:\input.dat" );

-- Loop from 1 to number of GPAs
for Index in 1 .. Num_GPAs loop

    --      Read current GPA from Input_File
    Get ( File => Input_File,
          Item => GPAs(Index) );
    Skip_Line (File => Input_File);

end loop;

-- Close Input_File
Close ( File => Input_File );

```

We've assumed that there's exactly one GPA per line in the input file; that's why we have a `Skip_Line` after each `Get` from the file. In general, you'll need to know the format of your input file to read from it properly.

So there you have it -- reading from a file isn't much harder than reading from the keyboard, but it makes things a LOT more convenient when we're dealing with large amounts of data.

11.5. End_of_File and End_of_Line

In the example above, we knew exactly how many GPAs were in the file. But what if we didn't know exactly how big the file was -- could we just read from the file until we got to the end, then stop? The answer is yes (big surprise there!). The `Ada.Text_IO` package provides a function called `End_Of_File` that returns `True` if we're at the end of the file and `False` if we're not. Of course, if we're reading characters from the input file we might not know how long each line in the file is either. We can tell when we get to the end of a particular line by using another `Ada.Text_IO` function called `End_Of_Line`. This function returns `True` if we're at the end of a line in the file and `False` otherwise. Let's try an example that uses both of these functions.

Example 11.2. Echoing a File to the Screen

Problem Description: Read in all the characters in a file called `c:\input.dat`, echoing each character to the screen as it's read in. Assume `Input_File` has been declared as `File_Type` and `Current_Character` has been declared as a `Character`, but we don't know how many lines are in the file or the length of each line.

Here's one algorithm that should work:

```
-- Open Input_File
-- While we're not at the end of the file
--   While we're not at the end of the line
--     Read in a character from the file
--     Print the character to the screen
--   Skip to the next line in the file
--   Move to a new line on the screen
-- Close Input_File
```

And the resulting code:

```
-- Open Input_File
Open ( File => Input_File,
      Mode => In_File,
      Name => "c:\input.dat" );

-- While we're not at the end of the file
while not End_Of_File (File => Input_File) loop

    -- While we're not at the end of the line
    while not End_Of_Line (File => Input_File) loop

        -- Read in a character from the file
        Get (File => Input_File,
             Item => Current_Character);

        -- Print the character to the screen
        Put (Item => Current_Character);

    end loop;

end loop;
```

```

--      Skip to the next line in the file
Skip_Line (File => Input_File);

--      Move to a new line on the screen
New_Line;

end loop;

-- Close Input_File
Close ( File => Input_File );

```

So now we can read from an input file whether or not we know how large the file is.

11.6. Creating a File for Writing

We mentioned earlier that files can also be used for output. Instead of printing data to the screen, we can print the data to a file that we can view with an editor or that some other program can use as input data. Instead of opening the file for reading, we simply create the file for writing. The syntax is as follows:

Creating Files

```

Create ( File => <variable name>,
        Mode => Out_File,
        Name => "<external file name>" );

```

<variable name> - the name of the file variable.

<external file name> - the name of the external file on the disk.

You've almost certainly noticed the similarities between `Open` and `Create`. The only differences are that when we create a file we use `Out_File` as the mode (since we're using the file for output) and that `Open` assumes the file already exists while `Create` doesn't make that assumption. You still need to be careful, though, because if the file you're creating already does exist, the `Create` will erase the contents of that file!

So let's create a file called `c:\output.dat` for writing, with the file variable `Output_File`:

```
Create (File => Output_File,
        Mode => Out_File,
        Name => "c:\output.dat");
```

Pretty easy, huh?

11.7. Printing Output to a File

To print to the file once it's created, we use the `Put` procedure we've been using all along; we simply need to make the output go to the file instead of to the screen. Let's say we wanted to print the first GPA in our array to a file. We'd tell the computer to use the file (instead of the screen) for output by using:

```
Put (File => Output_File,
     Item => GPAs(1),
     Fore => 1,
     Aft  => 2,
     Exp  => 0);
```

and if we want to move to the next line in the output file, we simply use:

```
New_Line (File => Output_File);
```

And that's all there is to it!

11.8. Putting It All Together

Let's extend the problem at the end of the last chapter to use file IO. Here's the revised problem description:

Read in a set of 50 test scores from a file called `c:\scores.dat` and calculate the average. Also determine what the highest and lowest scores in the set are. Print the average and high and low scores to a file called `c:\stats.dat`.

Understand the Problem

The problem is essentially the same as before, except now we need to use files for input and output (you sure don't want to type in 50 scores!).

Design a Solution

The high-level algorithm for our solution stays the same:

```
-- Print introduction
-- Read in scores
-- Calculate Average
-- Find high score
-- Find low score
-- Print out average, high score, and low score
```

as do our subproblems:

```
Print Introduction
Get Scores
Calculate Average
Find High Score
Find Low Score
Print Average, High Score, and Low Score
```

Most of our comment blocks stay the same as before, with changes to the `Get_Scores` and `Print_Average_High_Low` procedures:

```
-----
--
-- Name : Print_Introduction
-- Description : This procedure prints the introduction
--               for the program
-- Algorithm :
--               Print the introduction
--
-----
```

```
-----  
--  
-- Name : Get_Scores  
-- Description : This procedure gets the scores from  
--               the input file  
-- Algorithm :  
--   Print reading from file message  
--   Open file for input  
--   Loop from 1 to number of scores  
--     Read in score from input file  
--   Close the input file  
--  
-----
```

```
-----  
--  
-- Name : Calculate_Average  
-- Description : This procedure calculates the average  
--               of the test scores  
-- Algorithm :  
--   Initialize Sum to 0  
--   Loop from 1 to number of scores  
--     Set Sum to Sum + current score  
--   Set Average to Sum / number of scores  
--  
-----
```

```
-----  
--  
-- Name : Find_High_Score  
-- Description : This procedure finds the highest test score  
-- Algorithm :  
--   Initialize High_Score to first score in array  
--   Loop from 2 to number of scores  
--     If current score is greater than High_Score  
--       Set High_Score to current score  
--  
-----
```

```
-----  
--  
-- Name : Find_Low_Score  
-- Description : This procedure finds the lowest test score
```



```

-- Algorithm :
--   Initialize Low_Score to first score in array
--   Loop from 2 to number of scores
--       If current score is less than Low_Score
--           Set Low_Score to current score
--
-----
--
-- Name : Print_Average_High_Low
-- Description : This procedure prints the average score,
--               the high score, and the low score to the output file
-- Algorithm :
--   Print writing data to file message
--   Create the file for output
--   Print the average score to the output file
--   Print the high score to the output file
--   Print the low score to the output file
--   Close the output file
--
-----

```

The information that goes into and out of each procedure stays the same as before. Although Ada lets us pass file variables around as parameters if we need (or want) to, there's no need to for this problem. Because `Get_Scores` is the only procedure to use the input file and `Print_Average_High_Low` is the only procedure to use the output file, we'll simply declare the file variables as local variables in those procedures. Our decomposition diagram is still as shown on the following page, and our procedure headers are still:

```

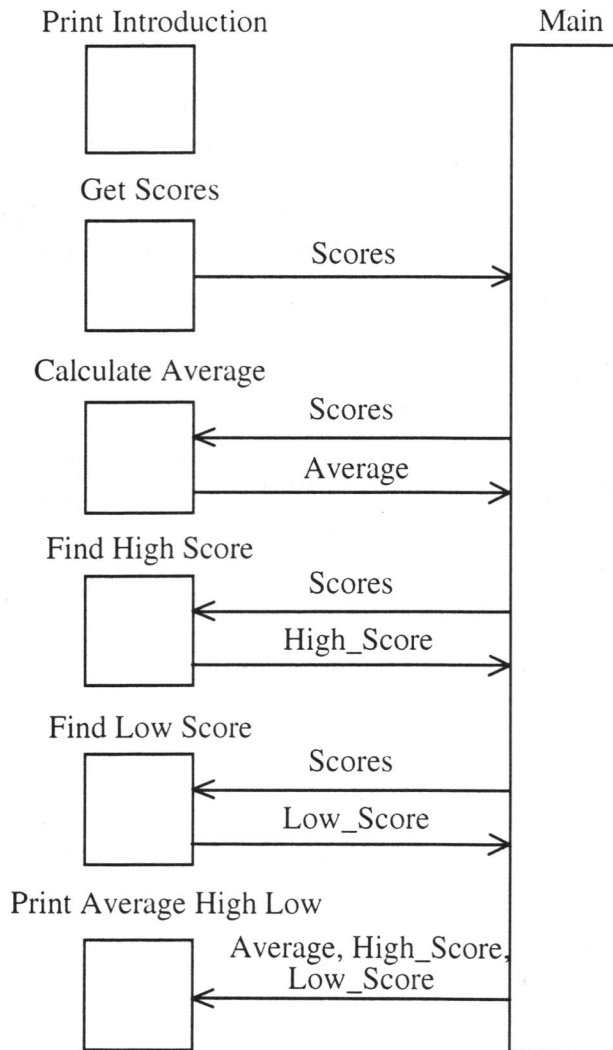
procedure Print_Introduction is

procedure Get_Scores ( Scores : out Score_Array ) is

procedure Calculate_Average ( Scores : in Score_Array;
    Average : out Float ) is

procedure Find_High_Score ( Scores : in Score_Array;
    High_Score : out Integer ) is

```



```

procedure Find_Low_Score ( Scores : in Score_Array;
    Low_Score : out Integer ) is

```

```

procedure Print_Average_High_Low ( Average : in Float;
    High_Score : in Integer;
    Low_Score : in Integer ) is

```

Now we're ready to move on to our next step.

Write the Test Plan

We'd like to test this essentially the same way we tested the program at the end of the previous chapter, but we'll list each of the 50 scores as a single input step instead of including them each separately. It certainly seems tedious to list all 50 values, but for someone else to use our test plan to check our results, we need to specify EXACTLY what our input is. Here are the resulting test cases:

Test Case 1 : Highest Score Near Middle, Lowest Score in First Element

Step	Input	Expected Results	Actual Results
1	c:\scores.dat file with the following scores: 0, 1, 99, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91	Print 70.82 for average, 100 for high score, 0 for low score to c:\stats.dat file	

Test Case 2 : Highest Score in First Element, Lowest Score in Last Element

Step	Input	Expected Results	Actual Results
1	c:\scores.dat file with the following scores: 100, 1, 91, 99, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 0	Print 70.82 for average, 100 for high score, 0 for low score to c:\stats.dat file	

Test Case 3 : Highest Score in Last Element, Lowest Score Near Middle

Step	Input	Expected Results	Actual Results
1	c:\scores.dat file with the following scores: 50, 1, 91, 99, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 0, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 100	Print 70.82 for average, 100 for high score, 0 for low score to c:\stats.dat file	

Write the Code

Here's the completed code for the program:

```

-----
--
-- Author : Paul McCartney
-- Description : This program reads in a set of 50 test
-- scores from an input file and calculates the average.
-- It also determines what the highest and lowest scores
-- in the set are. It then prints the average, high,
-- and low scores to a file.

```

```

-- Algorithm :
--   Print introduction
--   Read in scores
--   Calculate Average
--   Find high score
--   Find low score
--   Print out average, high score, and low score
--
-----

with Ada.Text_IO;
use  Ada.Text_IO;

with Ada.Float_Text_IO;
use  Ada.Float_Text_IO;

with Ada.Integer_Text_IO;
use  Ada.Integer_Text_IO;

procedure Process_Scores is

  Num_Scores : constant Integer := 50;
  type Score_Array is array (1..Num_Scores) of Integer;

  -----
  --
  -- Name : Print_Introduction
  -- Description : This procedure prints the introduction
  --   for the program
  -- Algorithm :
  --   Print the introduction
  --
  -----

procedure Print_Introduction is
begin

  --   Print the introduction
  Put (Item => "This program reads in a set of 50 ");
  Put (Item => "test scores from an input");
  New_Line;
  Put (Item => "file and calculates the average.  ");
  Put (Item => "It also determines what the highest");

```

```

New_Line;
Put (Item => "and lowest scores in the set are. ");
Put (Item => "It then prints the average, high,");
New_Line;
Put (Item => "and low scores to a file.");
New_Line;
New_Line;

end Print_Introduction;

-----
--
-- Name : Get_Scores
-- Description : This procedure gets the scores from
--               the input file
-- Algorithm :
--   Print reading from file message
--   Open file for input
--   Loop from 1 to number of scores
--   Read in score from input file
--   Close the input file
--
-----

procedure Get_Scores ( Scores : out Score_Array ) is

    Input_File : File_Type;

begin

    --   Print reading from file message
    Put_Line (Item => "Reading scores from file ...");

    --   Open file for input
    Open (File => Input_File,
          Mode => In_File,
          Name => "c:\scores.dat");

    --   Loop from 1 to number of scores
    for Index in 1 .. Num_Scores loop

```

```

        --      Read in score from input file
        Get (File => Input_File,
            Item => Scores(Index));
        Skip_Line (File => Input_File);

    end loop;

    --      Close the input file
    Close (File => Input_File);

end Get_Scores;

-----
--
-- Name : Calculate_Average
-- Description : This procedure calculates the average of
--      the test scores
-- Algorithm :
--      Initialize Sum to 0
--      Loop from 1 to number of scores
--          Set Sum to Sum + current score
--      Set Average to Sum / number of scores
--
-----

procedure Calculate_Average ( Scores : in Score_Array;
    Average : out Float ) is

    Sum : Integer;

begin

    --      Initialize Sum to 0
    Sum := 0;

    --      Loop from 1 to number of scores
    for Index in 1 .. Num_Scores loop

        --      Set Sum to Sum + current score
        Sum := Sum + Scores(Index);

    end loop;

```



```

--      Set Average to Sum / number of scores
Average := Float(Sum) / Float(Num_Scores);

end Calculate_Average;

-----
--
-- Name : Find_High_Score
-- Description : This procedure finds the highest test
--               score
-- Algorithm :
--   Initialize High_Score to first score in array
--   Loop from 2 to number of scores
--       If current score is greater than High_Score
--       Set High_Score to current score
--
-----

procedure Find_High_Score ( Scores : in Score_Array;
                           High_Score : out Integer ) is
begin
    --   Initialize High_Score to first score in array
    High_Score := Scores(1);

    --   Loop from 2 to number of scores
    for Index in 2 .. Num_Scores loop

        --       If current score is greater than
        --       High_Score
        if ( Scores(Index) > High_Score ) then

            --           Set High_Score to current score
            High_Score := Scores(Index);

        end if;

    end loop;

end Find_High_Score;

```

```

-----
--
-- Name : Find_Low_Score
-- Description : This procedure finds the lowest test
--               score
-- Algorithm :
--   Initialize Low_Score to first score in array
--   Loop from 2 to number of scores
--       If current score is less than Low_Score
--           Set Low_Score to current score
--
-----

procedure Find_Low_Score ( Scores : in Score_Array;
    Low_Score : out Integer ) is
begin

    --   Initialize Low_Score to first score in array
    Low_Score := Scores(1);

    --   Loop from 2 to number of scores
    for Index in 2 .. Num_Scores loop

        --       If current score is less than Low_Score
        if ( Scores(Index) < Low_Score ) then

            --           Set Low_Score to current score
            Low_Score := Scores(Index);

        end if;

    end loop;

end Find_Low_Score;

-----
--
-- Name : Print_Average_High_Low
-- Description : This procedure prints the average score,
--               the high score, and the low score to the output
--               file

```

```

-- Algorithm :
--   Print writing data to file message
--   Create the file for output
--   Print the average score to the output file
--   Print the high score to the output file
--   Print the low score to the output file
--   Close the output file
--
-----

procedure Print_Average_High_Low ( Average : in Float;
    High_Score : in Integer;
    Low_Score  : in Integer ) is

    Output_File : File_Type;

begin

    --   Print writing data to file message
    Put_Line (Item => "Writing data to file ...");

    --   Create the file for output
    Create (File => Output_File,
        Mode => Out_File,
        Name => "c:\stats.dat");

    --   Print the average score to the output file
    Put (File => Output_File,
        Item => "Average test score : ");
    Put (File => Output_File,
        Item => Average,
        Fore => 3,
        Aft  => 2,
        Exp  => 0);
    New_Line (File => Output_File);

    --   Print the high score to the output file
    Put (File => Output_File,
        Item => "High test score      : ");
    Put (File => Output_File,
        Item  => High_Score,
        Width => 3);
    New_Line (File => Output_File);

```

```

--      Print the low score to the output file
Put (File => Output_File,
    Item => "Low test score      : ");
Put (File => Output_File,
    Item  => Low_Score,
    Width => 3);
New_Line (File => Output_File);

--      Close the output file
Close (File => Output_File);

end Print_Average_High_Low;

```

```

-----
-- START OF MAIN PROGRAM
-----

```

```

Scores      : Score_Array;
Average      : Float;
High_Score   : Integer;
Low_Score    : Integer;

begin

--      Print introduction
Print_Introduction;

--      Read in scores
Get_Scores ( Scores => Scores );

--      Calculate Average
Calculate_Average ( Scores => Scores,
    Average => Average );

--      Find high score
Find_High_Score ( Scores => Scores,
    High_Score => High_Score);

--      Find low score
Find_Low_Score ( Scores => Scores,
    Low_Score => Low_Score);

```

```
--      Print out average, high score, and low score
Print_Average_High_Low ( Average => Average,
      High_Score => High_Score,
      Low_Score  => Low_Score);

end Process_Scores;
```

Test the Code

When we run the code, the results are as we expected:

Test Case 1 : Highest Score Near Middle, Lowest Score in First Element

Step	Input	Expected Results	Actual Results
1	c:\scores.dat file with the following scores: 0, 1, 99, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91	Print 70.82 for average, 100 for high score, 0 for low score to c:\stats.dat file	Prints 70.82 for average, 100 for high score, 0 for low score to c:\stats.dat file

Test Case 2 : Highest Score in First Element, Lowest Score in Last Element

Step	Input	Expected Results	Actual Results
1	c:\scores.dat file with the following scores: 100, 1, 91, 99, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 0	Print 70.82 for average, 100 for high score, 0 for low score to c:\stats.dat file	Prints 70.82 for average, 100 for high score, 0 for low score to c:\stats.dat file

Test Case 3 : Highest Score in Last Element, Lowest Score Near Middle

Step	Input	Expected Results	Actual Results
1	c:\scores.dat file with the following scores: 50, 1, 91, 99, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 0, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 100	Print 70.82 for average, 100 for high score, 0 for low score to c:\stats.dat file	Prints 70.82 for average, 100 for high score, 0 for low score to c:\stats.dat file

11.9. Common Mistakes

Confusing File and Name Parameters in Open/Create

Some people confuse their file variable name and the name of the file on their disk. Remember that the `File` => in the calls to `Open` and `Create` refers to the file variable, while the `Name` => in those calls refers to the name of the file on the disk. Getting these mixed up will result in a compilation error.

Trying to Read From an Out_File

This kind of error can occur if you've confused `Skip_Line` (which is for input) with `New_Line` (which is for output). If you try to do a `Skip_Line` in a file you're using for output, or a `New_Line` in a file you're using for input, your program will blow up during execution (with something called a `Mode_Error`). Make sure you only use input files for input and output files for output.

Trying to Open a Non-Existent File

If you haven't gotten the file name exactly right in the call to `Open`, your program will blow up with a `Name_Error`. This means that you've tried to open a file that doesn't exist. Carefully check to make sure you have the name right (and that the file exists). Sometimes it helps to actually specify the entire path name for the file rather than just the file name.

Forgetting the File Parameter in a Skip_Line or New_Line

If you want to do a `Skip_Line` from an input file, remember to include the file parameter; otherwise, the program will hang waiting for the user to press the <enter> key. Similarly, if you want to move to a new line in the output file, be sure to include the file parameter; otherwise, the new line goes to the screen, not the output file. If your program seems to hang for no apparent reason, or seems to print lots of extra blank lines to the screen, check for file parameters in your calls to `Skip_Line` and `New_Line`.

Trying to Read From a File That's Not Open

We always need to make sure that we've opened an input file before we try to read from it. Similarly, we always need to create our output file before we try to write to it. If we don't, our program will blow up with a `Status_Error`.

Trying to Read Past the End of a File

Unless we know exactly how many pieces of data are in our input file, we should use the `End_Of_File` function to make sure we don't read past the end of the file (if we do try to go past the end of a file, we get an `End_Error`). As a matter of fact, even if we DO know how much data is in the input file, we should still use `End_Of_File`; that way, if the input file is

changed to include more (or less) data, we don't have to change our program.

11.10. Exercises

1. Write a program that reads in 50 test percentages from a file and calculates the average test percentage.
2. Write a program that reads in 20 first initials, last names, and GPAs from a file. For each student, the program should determine whether or not that student is on Dean's List (GPA at least 3.0). The program should then print a list of the students on Dean's List to an output file.

Chapter 12. Putting It All Together

We've provided a section at the end of most of the chapters so far called "Putting It All Together". Hopefully, these sections helped solidify the concepts and constructs we were presenting in each chapter. In this chapter, we'll use the problem-solving skills we've developed and the Ada constructs we've learned to solve a larger, more complicated problem.

12.1. A Tic-Tac-Toe Game

Here's a problem description for a program that plays tic-tac-toe:

Play games of tic-tac-toe until the user says they want to quit. Each game consists of having the user and the computer alternate turns, with the tic-tac-toe board being displayed after each turn. After the game is over, the program should say whether the computer won, the user won, or the game was a tie.

Understand the Problem

Do we understand the problem? Who goes first? Let's let the user go first. How does the user indicate where they want their X or O to go? We'll number each of the spaces and have the user select the space they want by number. How should the computer decide where its O goes (we'll make the user X)? Let's have the computer go for the center first, then each of the corners, then finally the remaining spaces. Should the board be displayed graphically? That would be nice, but we'll just print out a 3 x 3 table of Xs and Os instead. To indicate empty spaces, we'll print the space number.

You may have gotten the impression in earlier chapters that this step is fairly trivial. Now that we're looking at a more complex example, you hopefully can see that this step isn't always that easy!

Design a Solution

To help us decide what the subproblems should be, let's write a high-level algorithm for our solution:

```

-- Print introduction
-- Initialize Play_Again to true
-- While Play_Again is true
    -- Initialize User_Won to false
    -- Initialize Computer_Won to false
    -- Initialize Board_Full to false
    -- Initialize Board to space numbers
    -- Display the board
    -- While User_Won, Computer_Won, and Board_Full are
    --     all false
    --     Let the user take a turn
    --     Display the board
    --     Check if user just won
    --     Check if board is full
    --     If User_Won and Board_Full are both false
    --         Let the computer take a turn
    --         Display the board
    --         Check if the computer won
    --         Check if board is full
    -- If User_Won is true
    --     Print user won message
    -- Otherwise, if Computer_Won is true
    --     Print computer won message
    -- Otherwise
    --     Print tie game message
    -- Ask user if they want to play again

```

Many of these steps look like good candidates for subproblems, but some of them seem simple enough to leave all the processing in the main program. Let's break the problem into the following subproblems:

- Print Introduction
- Initialize Board
- Display Board
- User Turn
- Computer Turn
- Check Won
- Check Full
- Get Play Again

Here are the comment blocks (including the detailed algorithms) for each of the procedures:

```

-----
--
-- Name : Print_Introduction
-- Description : This procedure prints the introduction
--               for the program
-- Algorithm :
--       Print the introduction
--
-----

-----
--
-- Name : Initialize_Board
-- Description : This procedure initializes each space of
--               the board to its space number
-- Algorithm :
--       Set Board(1,1) to '1'
--       Set Board(1,2) to '2'
--       Set Board(1,3) to '3'
--       Set Board(2,1) to '4'
--       Set Board(2,2) to '5'
--       Set Board(2,3) to '6'
--       Set Board(3,1) to '7'
--       Set Board(3,2) to '8'
--       Set Board(3,3) to '9'
--
-----

-----
--
-- Name : Display_Board
-- Description : This procedure displays the board
-- Algorithm :
--       Loop row from 1 to 3
--         Loop column from 1 to 3
--           Print Board(row,column)
--
-----

```

```

-----
--
-- Name : User_Turn
-- Description : This procedure lets the user take a turn
-- Algorithm :
--   Prompt for and get space number
--   While space number is < 1 or > 9
--     Print out of bounds message
--     Prompt for and get space number
--   Set Row to ( ( Space_Number - 1 ) / 3 ) + 1
--   Set Column to ( Space_Number rem 3 )
--   If Column = 0
--     Set Column to 3
--   While Board(Row,Column) is an X or O (space already
--     filled)
--     Print space filled message
--     Prompt for and get space number
--     While space number is < 1 or > 9
--       Print out of bounds message
--       Prompt for and get space number
--     Set Row to ( ( Space_Number - 1 ) / 3 ) + 1
--     Set Column to ( Space_Number rem 3 )
--     If Column = 0
--       Set Column to 3
--   Set Board(Row,Column) to X
--
-----

```

```

-----
--
-- Name : Computer_Turn
-- Description : This procedure lets the computer take a
--   turn. The computer tries for the middle first,
--   then each corner, then the remaining spaces.
-- Algorithm :
--   If Board(2,2) /= X and /= O
--     Set Board(2,2) to O
--   Otherwise, if Board(1,1) /= X and /= O
--     Set Board(1,1) to O
--   Otherwise, if Board(1,3) /= X and /= O
--     Set Board(1,3) to O
--   Otherwise, if Board(3,1) /= X and /= O
--     Set Board(3,1) to O
--
-----

```

```
--      Otherwise, if Board(3,3) /= X and /= O
--          Set Board(3,3) to O
--      Otherwise, if Board(1,2) /= X and /= O
--          Set Board(1,2) to O
--      Otherwise, if Board(2,3) /= X and /= O
--          Set Board(2,3) to O
--      Otherwise, if Board(3,2) /= X and /= O
--          Set Board(3,2) to O
--      Otherwise, if Board(2,1) /= X and /= O
--          Set Board(2,1) to O
--
```

```
-----
--
-- Name : Check_Won
-- Description : This procedure checks to see if the
--               player given by Who (X or O) has won
-- Algorithm :
--     Initialize Won to false
--     (Check rows for a win)
--     Loop row from 1 to 3
--         If Board(row,1) and Board(row,2) and
--         Board(row,3) = Who
--             Set Won to true
--     (Check columns for a win)
--     Loop column from 1 to 3
--         If Board(1,column) and Board(2,column) and
--         Board(3,column) = Who
--             Set Won to true
--     (Check diagonals)
--     If Board(1,1) and Board(2,2) and Board(3,3) = Who
--         Set Won to true
--     If Board(1,3) and Board(2,2) and Board(3,1) = Who
--         Set Won to true
--
```

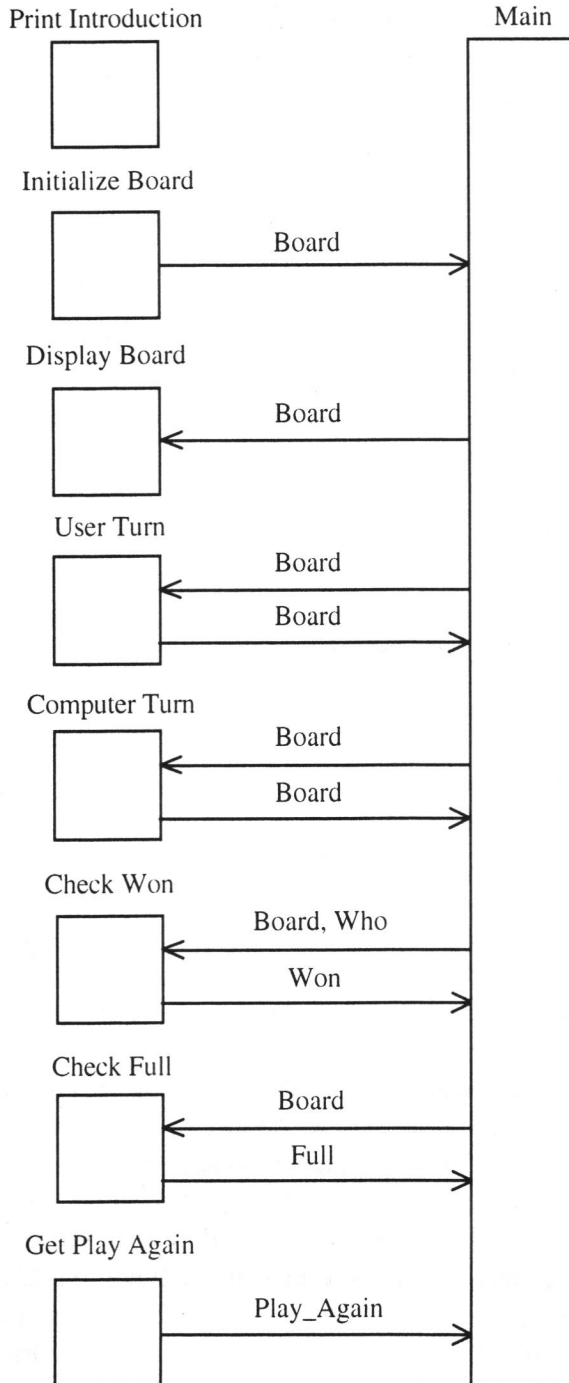
```
-----
--
-- Name : Check_Full
-- Description : This procedure checks to see if the board
--               is full
```

```

-- Algorithm :
--   Initialize Full to true
--   Loop row from 1 to 3
--       Loop column from 1 to 3
--           If Board(row,column) /= X and /= O
--               Set Full to false
--
-----
--
-- Name : Get_Play_Again
-- Description : This procedure finds out whether or not
--               the user wants to play again
-- Algorithm :
--   Prompt for and get Answer
--   While Answer is not n, N, y, or Y
--       Print error message
--       Prompt for and get Answer
--   If Answer is y or Y
--       Set Play_Again to true
--   Otherwise,
--       Set Play_Again to false
--
-----

```

The Print_Introduction procedure doesn't need any parameters. The Initialize_Board procedure initializes the board then passes it out to the main program. The Display_Board procedure simply displays the board, so it needs the board passed in from the main program. The User_Turn procedure looks at the board (to check for a valid move) and changes the board (once a valid move has been selected), so the board goes both in and out of this procedure. The same applies for the Computer_Turn procedure. The Check_Won procedure needs to look at the board and decide if who won, so both the board and the value of who come in to the procedure. The procedure then passes Won back out (Won is true if who won, false otherwise). The Check_Full procedure needs the board passed in (so it can check if it's full), then it passes Full out (Full is true if board is full, false otherwise). Finally, the Get_Play_Again procedures finds out



if the user wants to play again and passes out true if they do and false if they don't. The resulting decomposition diagram is on the previous page.

Here are the resulting procedure headers:

```

procedure Print_Introduction is

procedure Initialize_Board ( Board : out Board_Array ) is

procedure Display_Board ( Board : in Board_Array ) is

procedure User_Turn ( Board : in out Board_Array ) is

procedure Computer_Turn ( Board : in out Board_Array ) is

procedure Check_Won ( Board : in      Board_Array;
    Who : in      Character;
    Won :      out Boolean ) is

procedure Check_Full ( Board : in      Board_Array;
    Full :      out Boolean ) is

procedure Get_Play_Again ( Play_Again : out Boolean ) is

```

Now we're ready to move on to our test plan.

Write the Test Plan

The test plan for this code is fairly complicated. That's because there are three loops in the `User_Turn` procedure, one of which is nested inside another, as well as a loop in the `Get_Play_Again` procedure, a loop in the main program, and numerous `if` statements throughout the program. In addition, we need to play at least one game in which the user wins, one game in which the computer wins, and one game that results in a tie (to make sure we test all the branches of the last `if` in the main program). We also need to be sure to play a game in which the win occurs in a row, a game in which the win occurs in a column, and games in which the wins occur on each diagonal. Finally, we should probably check to see if the program works properly when the user wins by filling the last square on the board. Note that, with the user going first, the computer can't fill the last square on the board. Because the test cases are fairly complicated, we've

included tables after the test cases to show which test cases and steps test each construct.

Test Case 1

Step	Input	Expected Results	Actual Results
1	1 for space	Computer pick space 5	
2	0 for space	Error message, reprompt	
3	7 for space	Computer pick space 3	
4	1 for space	Error message, reprompt	
5	4 for space	User Won message	
6	y for play again	Prompt for space number	
7	10 for space	Error message, reprompt	
8	0 for space	Error message, reprompt	
9	4 for space	Computer pick space 5	
10	2 for space	Computer pick space 1	
11	4 for space	Error message, reprompt	
12	1 for space	Error message, reprompt	
13	6 for space	Computer pick space 3	
14	8 for space	Computer pick space 7 Computer Won message	
15	z for play again	Error message, reprompt	
16	q for play again	Error message, reprompt	
17	n for play again	Program ends	

Test Case 2

Step	Input	Expected Results	Actual Results
1	5 for space	Computer pick space 1	
2	7 for space	Computer pick space 3	
3	7 for space	Error message, reprompt	
4	0 for space	Error message, reprompt	
5	9 for space	Computer pick space 2 Computer Won message	
6	q for play again	Error message, reprompt	
7	Y for play again	Prompt for space number	
8	5 for space	Computer pick space 1	
9	3 for space	Computer pick space 7	
10	5 for space	Error message, reprompt	
11	10 for space	Error message, reprompt	
12	0 for space	Error message, reprompt	
13	9 for space	Computer pick space 2	
14	8 for space	Computer pick space 6	
15	4 for space	Tie Game message	
16	N for play again	Program ends	

Test Case 3

Step	Input	Expected Results	Actual Results
1	3 for space	Computer pick space 5	
2	7 for space	Computer pick space 1	
3	4 for space	Computer pick space 9 Computer Won message	
4	n for play again	Program ends	

Test Case 4

Step	Input	Expected Results	Actual Results
1	5 for space	Computer pick space 1	
2	6 for space	Computer pick space 3	
3	2 for space	Computer pick space 7	
4	9 for space	Computer pick space 8	
5	4 for space	User Won message	
6	n for play again	Program ends	

Test Case 4 plays a game in which the user wins by filling the last square on the board.

In the tables below, an entry like 1:2-3 indicates that Test Case 1, Steps 2 and 3 test the given construct (executing the loop body of the first loop in `User_Turn` 1 time, for example). Although a particular construct may be tested by many test case steps, we only reference the first steps that test the construct.

You should also note that in some cases testing a construct in a particular way is not possible. For example, we can't test the first loop in the main program 0 times because we initialize `Play_Again` to force the loop body to execute at least once.

Procedure : User_Turn

Loop	0 Times	1 Time	Multiple Times
1	1:1	1:2-3	1:7-9
2	1:1	1:4-5	1:11-13
3	1:5	2:1-5	2:10-13

If	False Branch	True Branch
1	1:1	2:9
2	1:5	1:13

Procedure : Computer_Turn

If	False Branch	True Branch
1	1:3	1:1
2	1:3	1:10
3	1:14	1:3
4	2:5	1:14
5	2:5	3:3
6	2:14	2:5
7	4:4	2:14
8	Not Possible	4:4

Procedure : Check_Won

If	False Branch	True Branch
1	1:1	2:5
2	1:1	1:5
3	1:1	3:3
4	1:1	1:14

Procedure : Check_Full

If	False Branch	True Branch
1	2:15	1:1

Procedure : Get_Play_Again

Loop	0 Times	1 Time	Multiple Times
1	1:6	2:6-7	1:15-17
If	False Branch	True Branch	
1	1:17	1:6	

Main Program

Loop	0 Times	1 Time	Multiple Times
1	Not Possible	3:1-4	1:1-17
2	Not Possible	Not Possible	1:1-17
If	False Branch	True Branch	
1	1:5	1:1	
2	1:14	1:5	
3	2:15	1:14	

Write the Code

When we write the (massive amounts of) code from our algorithms, we get:

```
-----
--
-- Author : Bob Seger
-- Description : This program plays games of tic-tac-toe
--               until the user says they want to quit. Each game
--               consists of having the user and the computer alternate
--               turns, with the tic-tac-board being displayed after
--               each turn. After the game is over, the program says
--               whether the computer won, the user won, or the game
--               was a tie.
-- Algorithm :
--   Print introduction
--   Initialize Play_Again to true
--   While Play_Again is true
--     Initialize User_Won to false
--     Initialize Computer_Won to false
```

```

--      Initialize Board_Full to false
--      Initialize Board to space numbers
--      Display the board
--      While User_Won, Computer_Won, and Board_Full are
--          all false
--          Let the user take a turn
--          Display the board
--          Check if user just won
--          Check if board is full
--          If User_Won and Board_Full are both false
--              Let the computer take a turn
--              Display the board
--              Check if the computer won
--              Check if board is full
--          If User_Won is true
--              Print user won message
--          Otherwise, if Computer_Won is true
--              Print computer won message
--          Otherwise
--              Print tie game message
--      Ask user if they want to play again
--

```

```

-----
with Ada.Text_IO;
use  Ada.Text_IO;

```

```

with Ada.Integer_Text_IO;
use  Ada.Integer_Text_IO;

```

```

procedure Tic_Tac_Toe is

```

```

    type Board_Array is array (1..3, 1..3) of Character;

```

```

-----
--
-- Name : Print_Introduction
-- Description : This procedure prints the introduction
--               for the program
-- Algorithm :
--               Print the introduction
--
-----

```

```
procedure Print_Introduction is
begin
```

```
    --      Print the introduction
    Put (Item => "This program plays games of ");
    Put (Item => "tic-tac-toe until the user");
    New_Line;
    Put (Item => "says they want to quit.  Each ");
    Put (Item => "game consists of having the");
    New_Line;
    Put (Item => "user and the computer alternate ");
    Put (Item => "turns, with the tic-tac-toe");
    New_Line;
    Put (Item => "board being displayed after each ");
    Put (Item => "turn.  After the game is");
    New_Line;
    Put (Item => "over, the program says whether ");
    Put (Item => "the computer won, the user");
    New_Line;
    Put (Item => "won, or the game was a tie.");
    New_Line;
```

```
end Print_Introduction;
```

```
-----
--
-- Name : Initialize_Board
-- Description : This procedure initializes each space of
-- the board to its space number
-- Algorithm :
--   Set Board(1,1) to '1'
--   Set Board(1,2) to '2'
--   Set Board(1,3) to '3'
--   Set Board(2,1) to '4'
--   Set Board(2,2) to '5'
--   Set Board(2,3) to '6'
--   Set Board(3,1) to '7'
--   Set Board(3,2) to '8'
--   Set Board(3,3) to '9'
--
-----
```



```
procedure Initialize_Board ( Board : out Board_Array ) is
begin
```

```
    -- Set Board(1,1) to '1'
    Board(1,1) := '1';
```

```
    -- Set Board(1,2) to '2'
    Board(1,2) := '2';
```

```
    -- Set Board(1,3) to '3'
    Board(1,3) := '3';
```

```
    -- Set Board(2,1) to '4'
    Board(2,1) := '4';
```

```
    -- Set Board(2,2) to '5'
    Board(2,2) := '5';
```

```
    -- Set Board(2,3) to '6'
    Board(2,3) := '6';
```

```
    -- Set Board(3,1) to '7'
    Board(3,1) := '7';
```

```
    -- Set Board(3,2) to '8'
    Board(3,2) := '8';
```

```
    -- Set Board(3,3) to '9'
    Board(3,3) := '9';
```

```
end Initialize_Board;
```

```
-----
--
-- Name : Display_Board
-- Description : This procedure displays the board
-- Algorithm :
--     Loop row from 1 to 3
--         Loop column from 1 to 3
--             Print Board(row,column)
--
-----
```

```
procedure Display_Board ( Board : in Board_Array ) is
begin
```

```
    New_Line;
```

```
    --      Loop row from 1 to 3
    for Row in 1 .. 3 loop
```

```
        --      Loop column from 1 to 3
        for Column in 1 .. 3 loop
```

```
            --      Print Board(row,column)
            Put (Item => "  ");
            Put (Item => Board(Row,Column));
```

```
        end loop;
```

```
    New_Line;
```

```
end loop;
```

```
    New_Line;
```

```
end Display_Board;
```

```
-----
--
-- Name : User_Turn
-- Description : This procedure lets the user take a turn
-- Algorithm :
--   Prompt for and get space number
--   While space number is < 1 or > 9
--     Print out of bounds message
--     Prompt for and get space number
--   Set Row to ( ( Space_Number - 1 ) / 3 ) + 1
--   Set Column to ( Space_Number rem 3 )
--   If Column = 0
--     Set Column to 3
--   While Board(Row,Column) is an X or O (space already
--     filled)
--     Print space filled message
--     Prompt for and get space number
--     While space number is < 1 or > 9
```

```

--          Print out of bounds message
--          Prompt for and get space number
--          Set Row to ( ( Space_Number - 1 ) / 3 ) + 1
--          Set Column to ( Space_Number rem 3 )
--          If Column = 0
--              Set Column to 3
--          Set Board(Row,Column) to X
--
-----

```

```

procedure User_Turn ( Board : in out Board_Array ) is

```

```

    Space_Number : Integer;
    Row          : Integer;
    Column       : Integer;

```

```

begin

```

```

    --          Prompt for and get space number
    Put (Item => "Please enter selected space number : ");
    Get (Item => Space_Number);
    Skip_Line;

```

```

    --          While space number is < 1 or > 9
    while ( Space_Number < 1 ) or
        ( Space_Number > 9 ) loop

```

```

        --          Print out of bounds message
        Put (Item => "Space number must be between ");
        Put (Item => "1 and 9 !!!");
        New_Line;

```

```

        --          Prompt for and get space number
        Put (Item => "Please re-enter selected space ");
        Put (Item => "number : ");
        Get (Item => Space_Number);
        Skip_Line;

```

```

    end loop;

```

```

    --          Set Row to ( ( Space_Number - 1 ) / 3 ) + 1
    Row := ( ( Space_Number - 1 ) / 3 ) + 1;

```

```

--      Set Column to ( Space_Number rem 3 )
Column := Space_Number rem 3;

--      If Column = 0
if ( Column = 0 ) then

    --          Set Column to 3
    Column := 3;

end if;

--      While Board(Row,Column) is an X or O
--          (space already filled)
while ( Board(Row,Column) = 'X' ) or
      ( Board(Row,Column) = 'O' ) loop

    --          Print space filled message
    Put (Item => "That space is already ");
    Put (Item => "filled !!!");
    New_Line;

    --      Prompt for and get space number
    Put (Item => "Please enter selected space ");
    Put (Item => "number : ");
    Get (Item => Space_Number);
    Skip_Line;

    --      While space number is < 1 or > 9
    while ( Space_Number < 1 ) or
          ( Space_Number > 9 ) loop

        --          Print out of bounds message
        Put (Item => "Space number must be between ");
        Put (Item => "1 and 9 !!!");
        New_Line;

        --          Prompt for and get space number
        Put (Item => "Please re-enter selected space ");
        Put (Item => "number : ");
        Get (Item => Space_Number);
        Skip_Line;

    end loop;

```

```

--      Set Row to ( ( Space_Number - 1 ) / 3 ) + 1
Row := ( ( Space_Number - 1 ) / 3 ) + 1;

--      Set Column to ( Space_Number rem 3 )
Column := Space_Number rem 3;

--      If Column = 0
if ( Column = 0 ) then

    --          Set Column to 3
    Column := 3;

end if;

end loop;

--      Set Board(Row,Column) to X
Board(Row,Column) := 'X';

end User_Turn;

-----
--
-- Name : Computer_Turn
-- Description : This procedure lets the computer take a
--               turn. The computer tries for the middle first,
--               then each corner, then the remaining spaces.
-- Algorithm :
--   If Board(2,2) /= X or O
--       Set Board(2,2) to O
--   Otherwise, if Board(1,1) /= X or O
--       Set Board(1,1) to O
--   Otherwise, if Board(1,3) /= X or O
--       Set Board(1,3) to O
--   Otherwise, if Board(3,1) /= X or O
--       Set Board(3,1) to O
--   Otherwise, if Board(3,3) /= X or O
--       Set Board(3,3) to O
--   Otherwise, if Board(1,2) /= X or O
--       Set Board(1,2) to O
--   Otherwise, if Board(2,3) /= X or O
--       Set Board(2,3) to O

```

```
--      Otherwise, if Board(3,2) /= X or O
--          Set Board(3,2) to O
--      Otherwise, if Board(2,1) /= X or O
--          Set Board(2,1) to O
--
```

```
-----
procedure Computer_Turn ( Board : in out Board_Array ) is
begin
```

```
    --      If Board(2,2) /= X or O
    if ( Board(2,2) /= 'X' ) and
        ( Board(2,2) /= 'O' ) then

        --          Set Board(2,2) to O
        Board(2,2) := 'O';

    --      Otherwise, if Board(1,1) /= X or O
    elsif ( Board(1,1) /= 'X' ) and
        ( Board(1,1) /= 'O' ) then

        --          Set Board(1,1) to O
        Board(1,1) := 'O';

    --      Otherwise, if Board(1,3) /= X or O
    elsif ( Board(1,3) /= 'X' ) and
        ( Board(1,3) /= 'O' ) then

        --          Set Board(1,3) to O
        Board(1,3) := 'O';

    --      Otherwise, if Board(3,1) /= X or O
    elsif ( Board(3,1) /= 'X' ) and
        ( Board(3,1) /= 'O' ) then

        --          Set Board(3,1) to O
        Board(3,1) := 'O';

    --      Otherwise, if Board(3,3) /= X or O
    elsif ( Board(3,3) /= 'X' ) and
        ( Board(3,3) /= 'O' ) then
```

```

--      Set Board(3,3) to O
Board(3,3) := 'O';

--      Otherwise, if Board(1,2) /= X or O
elsif ( Board(1,2) /= 'X' ) and
      ( Board(1,2) /= 'O' ) then

--      Set Board(1,2) to O
Board(1,2) := 'O';

--      Otherwise, if Board(2,3) /= X or O
elsif ( Board(2,3) /= 'X' ) and
      ( Board(2,3) /= 'O' ) then

--      Set Board(2,3) to O
Board(2,3) := 'O';

--      Otherwise, if Board(3,2) /= X or O
elsif ( Board(3,2) /= 'X' ) and
      ( Board(3,2) /= 'O' ) then

--      Set Board(3,2) to O
Board(3,2) := 'O';

--      Otherwise, if Board(2,1) /= X or O
elsif ( Board(2,1) /= 'X' ) and
      ( Board(2,1) /= 'O' ) then

--      Set Board(2,1) to O
Board(2,1) := 'O';

end if;

end Computer_Turn;

-----
--
-- Name : Check_Won
-- Description : This procedure checks to see if the
--               player given by Who (X or O) has won
-- Algorithm :
--   Initialize Won to false
--   (Check rows for a win)

```

```

--      Loop row from 1 to 3
--          If Board(row,1) and Board(row,2) and
--              Board(row,3) = Who
--              Set Won to true
--      (Check columns for a win)
--      Loop column from 1 to 3
--          If Board(1,column) and Board(2,column) and
--              Board(3,column) = Who
--              Set Won to true
--      (Check diagonals)
--      If Board(1,1) and Board(2,2) and Board(3,3) = Who
--          Set Won to true
--      If Board(1,3) and Board(2,2) and Board(3,1) = Who
--          Set Won to true
--
-----

```

```

procedure Check_Won ( Board : in      Board_Array;
    Who : in      Character;
    Won :      out Boolean ) is
begin
    --      Initialize Won to false
    Won := false;

    --      (Check rows for a win)
    --      Loop row from 1 to 3
    for Row in 1 .. 3 loop

        --          If Board(row,1) and Board(row,2) and
        --              Board(row,3) = Who
        if ( Board(Row,1) = Who ) and
            ( Board(Row,2) = Who ) and
            ( Board(Row,3) = Who ) then

            --              Set Won to true
            Won := true;

        end if;

    end loop;

```



```

--      (Check columns for a win)
--      Loop column from 1 to 3
for Column in 1 .. 3 loop

    --          If Board(1,column) and Board(2,column) and
    --          Board(3,column) = Who
    if ( Board(1,Column) = Who ) and
        ( Board(2,Column) = Who ) and
        ( Board(3,Column) = Who ) then

        --          Set Won to true
        Won := true;

    end if;

end loop;

--      (Check diagonals)
--      If Board(1,1) and Board(2,2) and
--      Board(3,3) = Who
if ( Board(1,1) = Who ) and
    ( Board(2,2) = Who ) and
    ( Board(3,3) = Who ) then

    --          Set Won to true
    Won := true;

end if;

--      If Board(1,3) and Board(2,2) and
--      Board(3,1) = Who
if ( Board(1,3) = Who ) and
    ( Board(2,2) = Who ) and
    ( Board(3,1) = Who ) then

    --          Set Won to true
    Won := true;

end if;

end Check_Won;

```

```

-----
--
-- Name : Check_Full
-- Description : This procedure checks to see if the
--               board is full
-- Algorithm :
--   Initialize Full to true
--   Loop row from 1 to 3
--     Loop column from 1 to 3
--       If Board(row,column) /= X and /= O
--         Set Full to false
--
-----

```

```

procedure Check_Full ( Board : in      Board_Array;
                       Full :    out Boolean ) is

```

```

begin

```

```

    --   Initialize Full to true
    Full := True;

    --   Loop row from 1 to 3
    for Row in 1 .. 3 loop

        --       Loop column from 1 to 3
        for Column in 1 .. 3 loop

            --           If Board(row,column) /= X and /= O
            if ( Board(Row,Column) /= 'X' ) and
               ( Board(Row,Column) /= 'O' ) then

                --               Set Full to false
                Full := False;

            end if;

        end loop;

    end loop;

end Check_Full;

```

```

-----
--
-- Name : Get_Play_Again
-- Description : This procedure finds out whether or not
--               the user wants to play again
-- Algorithm :
--   Prompt for and get Answer
--   While Answer is not n, N, y, or Y
--     Print error message
--     Prompt for and get Answer
--   If Answer is y or Y
--     Set Play_Again to true
--   Otherwise,
--     Set Play_Again to false
--
-----

```

```

procedure Get_Play_Again ( Play_Again : out Boolean ) is

```

```

    Answer : Character;

```

```

begin

```

```

    -- Prompt for and get Answer
    Put (Item => "Would you like to play again (y,n)? ");
    Get (Item => Answer);
    Skip_Line;

```

```

    -- While Answer is not n, N, y, or Y
    while ( Answer /= 'n' ) and ( Answer /= 'N' ) and
        ( Answer /= 'y' ) and ( Answer /= 'Y' ) loop

```

```

        -- Print error message
        Put_Line (Item => "Answer must be n, N, y, or Y!");

```

```

        -- Prompt for and get Answer
        Put (Item => "Would you like to play again ");
        Put (Item => "(y,n)? ");
        Get (Item => Answer);
        Skip_Line;

```

```

    end loop;

```

```

--      If Answer is y or Y
if ( Answer = 'y' ) or ( Answer = 'Y' ) then

    --      Set Play_Again to true
    Play_Again := True;

--      Otherwise,
else

    --      Set Play_Again to false
    Play_Again := False;

end if;

end Get_Play_Again;

```

```

-----
--  START OF MAIN PROGRAM
-----

```

```

Board      : Board_Array;
User_Won   : Boolean;
Computer_Won : Boolean;
Board_Full  : Boolean;
Play_Again  : Boolean;

```

```

begin

```

```

--      Print introduction
Print_Introduction;

```

```

--      Initialize Play_Again to true
Play_Again := True;

```

```

--      While Play_Again is true
while Play_Again loop

```

```

--          Initialize User_Won to false
User_Won := False;

```

```

--          Initialize Computer_Won to false
Computer_Won := False;

```

```

--      Initialize Board_Full to false
Board_Full := False;

--      Initialize Board to space numbers
Initialize_Board (Board => Board);

--      Display the board
Display_Board (Board => Board);

--      While User_Won, Computer_Won, and Board_Full
--      are all false
while ( not User_Won) and ( not Computer_Won )
    and ( not Board_Full ) loop

    --      Let the user take a turn
    User_Turn (Board => Board);

    --      Display the board
    Display_Board (Board => Board);

    --      Check if user just won
    Check_Won (Board => Board,
        Who => 'X',
        Won => User_Won);

    --      Check if board is full
    Check_Full (Board => Board,
        Full => Board_Full);

    --      If User_Won and Board_Full are both
    --      false
    if ( not User_Won ) and ( not Board_Full ) then

        --      Let the computer take a turn
        Put_Line (Item => "Computer taking turn ... ");
        Computer_Turn (Board => Board);

        --      Display the board
        Display_Board (Board => Board);

```

```

--          Check if the computer won
Check_Won (Board => Board,
           Who => 'O',
           Won => Computer_Won);

--          Check if board is full
Check_Full (Board => Board,
           Full => Board_Full);

end if;

end loop;

--          If User_Won is true
if User_Won then

    --          Print user won message
    Put_Line (Item => "You won the game !!!");
    New_Line;

--          Otherwise, if Computer_Won is true
elseif Computer_Won then

    --          Print computer won message
    Put_Line (Item => "The computer won !");
    New_Line;

--          Otherwise
else

    --          Print tie game message
    Put_Line (Item => "Tie game !");
    New_Line;

end if;

--          Ask user if they want to play again
Get_Play_Again (Play_Again => Play_Again);

end loop;

end Tic_Tac_Toe;

```

Test the Code

So now we need to run our test plan to make sure the program works. When we run the program above and fill in the actual results in our test plan, we get the following (exactly as we expected):

Test Case 1

Step	Input	Expected Results	Actual Results
1	1 for space	Computer pick space 5	Computer picks space 5
2	0 for space	Error message, reprompt	Error message, reprompts
3	7 for space	Computer pick space 3	Computer picks space 3
4	1 for space	Error message, reprompt	Error message, reprompts
5	4 for space	User Won message	User Won message
6	y for play again	Prompt for space number	Prompts for space number
7	10 for space	Error message, reprompt	Error message, reprompts
8	0 for space	Error message, reprompt	Error message, reprompts
9	4 for space	Computer pick space 5	Computer picks space 5
10	2 for space	Computer pick space 1	Computer picks space 1
11	4 for space	Error message, reprompt	Error message, reprompts
12	1 for space	Error message, reprompt	Error message, reprompts
13	6 for space	Computer pick space 3	Computer picks space 3
14	8 for space	Computer pick space 7 Computer Won message	Computer picks space 7 Computer Won message
15	z for play again	Error message, reprompt	Error message, reprompts

16	q for play again	Error message, reprompt	Error message, reprompts
17	n for play again	Program ends	Program ends

Test Case 2

Step	Input	Expected Results	Actual Results
1	5 for space	Computer pick space 1	Computer picks space 1
2	7 for space	Computer pick space 3	Computer picks space 3
3	7 for space	Error message, reprompt	Error message, reprompts
4	0 for space	Error message, reprompt	Error message, reprompts
5	9 for space	Computer pick space 2 Computer Won message	Computer picks space 2 Computer Won message
6	q for play again	Error message, reprompt	Error message, reprompts
7	Y for play again	Prompt for space number	Prompts for space number
8	5 for space	Computer pick space 1	Computer picks space 1
9	3 for space	Computer pick space 7	Computer picks space 7
10	5 for space	Error message, reprompt	Error message, reprompts
11	10 for space	Error message, reprompt	Error message, reprompts
12	0 for space	Error message, reprompt	Error message, reprompts
13	9 for space	Computer pick space 2	Computer picks space 2
14	8 for space	Computer pick space 6	Computer picks space 6
15	4 for space	Tie Game message	Tie Game message
16	N for play again	Program ends	Program ends

Test Case 3

Step	Input	Expected Results	Actual Results
1	3 for space	Computer pick space 5	Computer picks space 5
2	7 for space	Computer pick space 1	Computer picks space 1
3	4 for space	Computer pick space 9 Computer Won message	Computer picks space 9 Computer Won message
4	n for play again	Program ends	Program ends

Test Case 4

Step	Input	Expected Results	Actual Results
1	5 for space	Computer pick space 1	Computer picks space 1
2	6 for space	Computer pick space 3	Computer picks space 3
3	2 for space	Computer pick space 7	Computer picks space 7
4	9 for space	Computer pick space 8	Computer picks space 8
5	4 for space	User Won message	User Won message
6	n for play again	Program ends	Program ends

And there you have it! We applied our five step process to a reasonably large problem (at least at the introductory level) and managed to solve the problem just fine.

Chapter 13. Advanced Topics

Well, we've pretty much covered everything you need to successfully complete an introductory course. Some of you, however, would probably like to know more about some of the other constructs available in Ada. This chapter introduces some of those more advanced topics. Although it certainly doesn't cover the rest of Ada 95, it does give you enough information to explore Ada a little further.

13.1. Writing Your Own Packages

You hopefully already realize how valuable packages are. Without the `Ada.Text_IO` package, for example, you'd have to write your own input and output procedures for handling character and string I/O! The standard packages provided in Ada let you use the code someone else has already written for you. The other nice thing about packages is that all you have to do to use them is include a `with` (and `use`, if you like) clause at the top of your program to provide access to the "stuff" in that package.

But what if you want to write your own package? You might have a great idea for some procedures that other people might want to use, and you know the cleanest way to do it is by including them in a package. You'll need to write two things: a package specification, which defines the interface to the package (how you call each procedure, for example), and a package body, which provides the executable code for each procedure. A general view of the required syntax for a package specification is provided on the following page.

The only thing in the package specification that you haven't seen yet is *procedure specifications*. A procedure specification is simply the procedure header we've been writing all along, with the `is` at the end replaced by a semicolon. We'll look at an example in a moment.

Package Specification Syntax

```

<package comment block>

<withs and uses of other packages>

package <package name> is

    <constant declarations>

    <type definitions>

    <procedure specifications in the package>

end <package name>;

```

<package comment block> - the comment block for the package

<withs and uses of other packages> - with and use clauses to provide access to data types in other packages

<package name> - the name of the package

<constant declarations> - declarations of new constants for the package

<type definitions> - definitions of new data types for the package

<procedure specifications in the package> - specifications for the procedures in the package

The syntax for a package body is provided on the following page. The big differences between the package specification and the package body are that we include the word `body` in the package header for the package body, and we include the procedure bodies (essentially, the procedures we've been writing all along) rather than the procedure specifications in the package body.

Package Body Syntax

```

<package comment block>

<withs and uses of other packages>

package body <package name> is

    <constant declarations>

    <procedure bodies in the package>

end <package name>;

```

<package comment block> - the comment block for the package
 <withs and uses of other packages> - with and use clauses to provide
 access to data types and procedures in other packages
 <package name> - the name of the package
 <constant declarations> - declarations of new constants used in the
 package body
 <procedure bodies in the package> - bodies for the procedures in the
 package

OK, let's look at an example.

Example 13.1. Creating a Package for Vector Operations

Problem Description: Write a package that provides procedures to calculate the magnitude of a vector, the dot product of two vectors, and the cross product of two vectors.

Let's look at the package specification first; here's one that should work:

```

-----
--
-- Author : Gary Cherone
-- Description : This package provides a data type for
--   vectors and some basic vector operations.

```

--

package Vectors is

-- declare array to hold i, j, and k vector components
type Vector is array (1..3) of Float;

--

-- Name : Magnitude

-- Description : This procedure calculates the magnitude
-- of a vector.

--

procedure Magnitude (The_Vector : in Vector;
Mag : out Float);

--

-- Name : Dot

-- Description : This procedure calculates the dot
-- product of two vectors.

--

procedure Dot (Vector_1 : in Vector;
Vector_2 : in Vector;
Dot_Product : out Float);

--

-- Name : Cross

-- Description : This procedure calculates the cross
-- product of two vectors.

--

procedure Cross (Vector_1 : in Vector;
Vector_2 : in Vector;
Cross_Product : out Vector);

```
end Vectors;
```

There are several items worth noting here. One of the first things we did in the package specification is define a new data type, called `vector`, that's an array of three floating point numbers. The first element of the array is the *i* component of the vector, the second element is the *j* component, and the third element is the *k* component. Each procedure comment block in the package specification contains the name of the procedure and a description of the procedure, but no algorithm! That's because we're really interested in specifying the interface to the procedures when we write the package specification; we don't worry about how we're going to implement those procedures until we get to the package body. Finally, the procedure headers are slightly different from what we're used to seeing, since they end in semicolons rather than the word `is`.

Moving on to the package body, we could have something like:

```
-----
--
-- Author : Gary Cherone
-- Description : This package provides a data type for
--               vectors and some basic vector operations.
--
-----

with Ada.Numerics.Elementary_Functions;
use  Ada.Numerics.Elementary_Functions;

package body Vectors is

    -----
    --
    -- Name : Magnitude
    -- Description : This procedure calculates the magnitude
    --               of a vector.
    -- Algorithm :
    --               Calculate Mag as square root of i ** 2 + j ** 2 +
    --               k ** 2
    --
    -----
```

```

procedure Magnitude ( The_Vector : in      Vector;
                      Mag :      out Float ) is
begin
    --      Calculate Mag as square root of i ** 2 + j ** 2
    --      + k ** 2
    Mag := Sqrt ( The_Vector(1)**2 + The_Vector(2)**2 +
                  The_Vector(3)**2 );

end Magnitude;

-----
--
-- Name : Dot
-- Description : This procedure calculates the dot
--               product of two vectors.
-- Algorithm :
--   Calculate Dot_Product as Vector_1(1)*Vector_2(1) +
--   Vector_1(2)*Vector_2(2) +
--   Vector_1(3)*Vector_2(3)
--
-----

procedure Dot ( Vector_1 : in      Vector;
               Vector_2 : in      Vector;
               Dot_Product :      out Float ) is
begin
    --      Calculate Dot_Product as Vector_1(1)*Vector_2(1)
    --      + Vector_1(2)*Vector_2(2) +
    --      Vector_1(3)*Vector_2(3)
    Dot_Product := Vector_1(1) * Vector_2(1) +
                   Vector_1(2) * Vector_2(2) +
                   Vector_1(3) * Vector_2(3);

end Dot;

-----
--
-- Name : Cross
-- Description : This procedure calculates the cross
--               product of two vectors.

```

```

-- Algorithm :
--   Calculate Cross_Product(1) as
--       Vector_1(2)*Vector_2(3) -
--       Vector_1(3)*Vector_2(2)
--   Calculate Cross_Product(2) as
--       Vector_1(3)*Vector_2(1) -
--       Vector_1(1)*Vector_2(3)
--   Calculate Cross_Product(3) as
--       Vector_1(1)*Vector_2(2) -
--       Vector_1(2)*Vector_2(1)
--
-----

procedure Cross ( Vector_1 : in      Vector;
                  Vector_2  : in      Vector;
                  Cross_Product : out Vector ) is
begin

    --   Calculate Cross_Product(1) as
    --       Vector_1(2)*Vector_2(3) -
    --       Vector_1(3)*Vector_2(2)
    Cross_Product(1) := Vector_1(2)*Vector_2(3) -
        Vector_1(3)*Vector_2(2);

    --   Calculate Cross_Product(2) as
    --       Vector_1(3)*Vector_2(1) -
    --       Vector_1(1)*Vector_2(3)
    Cross_Product(2) := Vector_1(3)*Vector_2(1) -
        Vector_1(1)*Vector_2(3);

    --   Calculate Cross_Product(3) as
    --       Vector_1(1)*Vector_2(2) -
    --       Vector_1(2)*Vector_2(1)
    Cross_Product(3) := Vector_1(1)*Vector_2(2) -
        Vector_1(2)*Vector_2(1);

end Cross;

end Vectors;

```

Note that we don't have to define the `Vector` data type again in the package body; it's simply available to us from the package specification.

All the procedure comment blocks now contain algorithms, and each procedure has been fully implemented.

To use this package in a program, we would with and use the package:

```
with Vectors;
use Vectors;
```

declare variables of type Vector:

```
Position : Vector;
Force    : Vector;
Torque   : Vector;
```

and call the appropriate procedures in the `Vectors` package as needed; for example:

```
Cross ( Vector_1 => Position,
        Vector_2  => Force,
        Cross_Product => Torque );
```

That's how you write and use your own packages.

13.2. Enumeration Types

Given the data types we've seen in prior chapters, we can store all the data we'd need in the problems we'd see in an introductory course. There are times, however, when it would be more convenient to use *enumeration types*. An enumeration type is a data type for which we've explicitly enumerated all the possible values for a variable of that type. Let's look at an example to solidify this idea.

Example 13.2. Storing and Printing Music Preference

Problem Description: Write a code fragment that prints out a stored musical preference. Available preferences are Rock and Roll, Blues, Jazz, or Classical.

Without enumeration types, we'd probably use a character to store the musical preference (R for Rock and Roll, B for Blues, J for Jazz, and C for Classical). To print out the stored preference, we'd have to do something like:

```
-- If Preference is R
if ( Preference = 'R' ) then

    -- Print Rock and Roll
    Put (Item => "Preference : Rock and Roll");
    New_Line;

-- Otherwise, if Preference is B
elseif ( Preference = 'B' ) then

    -- Print Blues
    Put (Item => "Preference : Blues");
    New_Line;

-- Otherwise, if Preference is J
elseif ( Preference = 'J' ) then

    -- Print Jazz
    Put (Item => "Preference : Jazz");
    New_Line;

-- Otherwise,
else

    -- Print Classical
    Put (Item => "Preference : Classical");
    New_Line;

end if;
```

We can obviously make this work, but it seems pretty awkward. How can enumeration types help us? The first thing we'd do is define our new data type:

```
type Music_Type is (Rock_And_Roll, Blues, Jazz,
    Classical);
```

Note that we give the data type a name (`Music_Type`) and explicitly enumerate all the valid values for that data type (`Rock_And_Roll`, `Blues`, `Jazz`, and `Classical`). The general syntax for defining enumeration types is as follows:

Enumeration Type Definition Syntax

```
type <type name> is ( <valid value>, <valid value>,
    ... , <valid value> );
```

<type name> - the name of the new data type

<valid value> - one valid value for the new data type

Note : the ... in the above syntax is not included in the type definition; it simply indicates that the programmer picks how many valid values the new data type will have

Next, we make sure we have I/O capability for this data type (just as we did for Booleans way back in Chapter 6) by including the following immediately after the type definition:

```
package Music_Type_IO is new Ada.Text_IO Enumeration_IO (
    Enum => Music_Type);
use Music_Type_IO;
```

Finally, we can declare a variable of this type in the expected way:

```
Preference : Music_Type;
```

Now, to print the music preference, all we have to do is:

```
-- Print music preference
Put (Item => "Preference : ");
Put (Item => Preference);
New_Line;
```

By using an enumeration type, we've greatly simplified our code to output the musical preference.

13.3. Using Records

We've already learned about one data type that can store more than one piece of information -- arrays. Arrays are certainly useful (as we've seen), but sometimes they're not quite what we want. For example, say we want to hold information (Last Name, Age, GPA) about a single person in a single variable. We can't use an array of three elements, since all the elements of an array have to be the same data type. Instead, we use something called a *record*, which has a number of *fields*. The fields in a record can be of different data types, solving our problem for this example. Let's take a look at the syntax for a record type definition and variable declaration:

Record Type Definition and Variable Declaration

```
type <type name> is record
  <field name> : <field type>;
  <field name> : <field type>;
end record;

<variable name> : <type name>;
```

<type name> - the name of the new record type we're defining.

<field name> - the name of each field in the record.

<field type> - the data type for each field of the record.

<variable name> - the name of a variable declared as our record type.

Let's solidify this by defining the type and declaring a variable for our example:

```
type Info_Record is record
  Last_Name : String(1..15);
  Age       : Integer;
  GPA       : Float;
end record;
```

```
Person_Info : Info_Record;
```

The last thing we really need to know about records is how we access the fields of a record. Instead of indexing into a particular element, as we do with arrays, we use something called *dot notation* to get at a record's fields. In dot notation, we simply say <variable name>.<field name> to get at a particular field (note the dot between the variable name and the field name). So if we wanted to put a 35 into the Age field of the Person_Info variable, we would simply use:

```
Person_Info.Age := 35;
```

One other comment -- our example only stores information about a single person, but you'd probably want to store information about several people. Is there a way to store a bunch of these records? Sure there is -- an array! You can have an array of records, records with arrays as one or more of the fields, and just about any other combination you can think of!

Let's do an example:

Example 13.3. Reading in Information for 20 Students

Problem Description: Assuming the following data type definitions and variable declaration:

```
type Info_Record is record
  Last_Name : String(1..15);
  Age       : Integer;
  GPA       : Float;
end record;

Num_Students : constant Integer := 20;
type Info_Array is array (1..Num_Students) of
  Info_Record;

Student_Info : Info_Array;
```

write the code to read in the information for all 20 students.

Here's an algorithm that should work pretty well:

```
-- Loop from 1 to number of students
--   Prompt for and get last name
--   Prompt for and get age
--   Prompt for and get GPA
```

and turning this into code, we have:

```
-- Loop from 1 to number of students
for Index in 1 .. Num_Students loop

    --   Prompt for and get last name
    Put (Item => "Please enter last name for Student ");
    Put (Item => Index,
        Width => 1);
    Put (Item => " : ");
    Get (Item => Student_Info(Index).Last_Name);
    Skip_Line;

    --   Prompt for and get age
    Put (Item => "Please enter age for Student ");
    Put (Item => Index,
        Width => 1);
    Put (Item => " : ");
    Get (Item => Student_Info(Index).Age);
    Skip_Line;

    --   Prompt for and get GPA
    Put (Item => "Please enter GPA for Student ");
    Put (Item => Index,
        Width => 1);
    Put (Item => " (0.0-4.0) : ");
    Get (Item => Student_Info(Index).GPA);
    Skip_Line;

end loop;
```

Pay particularly close attention to the way we read into a specific field of the record found at a specific element in the array.

Hopefully you can see that combining records and arrays gives us a powerful tool for storing and using our program data.

13.4. Creating and Calling Functions

Functions are a lot like procedures; we call them with zero or more parameters, they do some work, then they terminate. The big difference is that procedures can send more than one piece of information back to the caller (by using more than one out or in out parameters). Functions, on the other hand, can only return a single value. That means that all the parameters for a function are in parameters, and the returned value is handled in a special way. Let's take a look at an example function:

```
-----
--
-- Name : Add
-- Description : This function adds two numbers and
--               returns the result
-- Algorithm :
--               Return the sum of the two numbers
--
-----

function Add (First_Number : in Integer;
              Second_Number : in Integer)
  return Integer is

begin

  --       Return the sum of the two numbers
  return First_Number + Second_Number;

end Add;
```

To call the function, we might do something like (assuming Number_1 and Number_2 are integer variables):

```
Sum := Add (First_Number => Number_1,
            Second_Number => Number_2);
```

Note that we call the function slightly differently from the way we call a procedure. Because the function returns a value, we can put the function call anywhere a value would be syntactically valid. We can't, therefore, simply say:

```
Add (First_Number => Number_1,
      Second_Number => Number_2);
```

because the computer needs a place to put the value returned from Add.

In general, everything that can be done with a function can be done with a procedure (but not vice versa). You can therefore solve your problems (at least the ones in this book) without ever using functions. Some people, however, prefer to use functions in those cases where the function calculates (or gets) and returns a single value.

13.5. Overloading Procedures and Functions

Whether you realize it or not, you've already been using overloaded procedures and functions! When we say a procedure (or function) is overloaded, we simply mean there's more than one procedure with the same name. For example, we've been using the `Put` procedure since Chapter 5, but there are actually a number of `Put` procedures (one for characters and strings, one for integers, one for floats, etc.). That means that the `Put` procedure is overloaded.

Let's use do an example to create our own overloaded procedures.

Example 13.4. Creating Several Min Procedures

Problem Description: Write overloaded `Min` procedures that work on Integers and Floats. Each procedure should determine the minimum of the two numbers.

The algorithms for the procedures will be identical:


```
-- If first number is less than second number
--   Set Minimum to first number
-- Otherwise,
--   Set Minimum to second number
```

The difference will be in the data types used in the procedure headers. Here are the full procedures:

```
-----
--
-- Name : Min
-- Description : This procedure the minimum of two
--               integers
-- Algorithm :
--   If first number is less than second number
--       Set Minimum to first number
--   Otherwise,
--       Set Minimum to second number
--
-----
```

```
procedure Min ( Number_1 : in      Integer;
                Number_2 : in      Integer;
                Minimum   : out Integer ) is
begin
    --   If first number is less than second number
    if ( Number_1 < Number_2 ) then
        --       Set Minimum to first number
        Minimum := Number_1;

        --   Otherwise,
    else
        --       Set Minimum to second number
        Minimum := Number_2;

    end if;

end Min;
```

```

-----
--
-- Name : Min
-- Description : This procedure the minimum of two
--               floats
-- Algorithm :
--   If first number is less than second number
--       Set Minimum to first number
--   Otherwise,
--       Set Minimum to second number
--
-----

```

```

procedure Min ( Number_1 : in      Float;
                Number_2 : in      Float;
                Minimum   : out Float ) is
begin
    --   If first number is less than second number
    if ( Number_1 < Number_2 ) then

        --       Set Minimum to first number
        Minimum := Number_1;

    --   Otherwise,
    else

        --       Set Minimum to second number
        Minimum := Number_2;

    end if;

end Min;

```

If we call Min using:

```

Min ( Number_1 => Integer_Variable_1,
      Number_2 => Integer_Variable_2,
      Minimum   => Integer_Variable_3);

```

we'll end up using the first Min procedure, while if we have:

```

Min ( Number_1 => Float_Variable_1,
      Number_2 => Float_Variable_2,
      Minimum  => Float_Variable_3);

```

we'll end up using the second `Min` procedure instead.

13.6. Using Exception Handlers

Most of you have probably already discovered that there are times when a program you wrote will "blow up" -- if you ask for a number and the user enters a character instead, if you try to open a file that doesn't exist, and so on. The program doesn't really blow up, of course; instead, Ada raises an *exception*, which then terminates the program. To keep our program from terminating because of the exception, we can handle that exception using an *exception handler*.

Let's look at an example. Say we have the following code:

```

-- Prompt for and get age
Put (Item => "Please enter age : ");
Get (Item => Age);
Skip_Line;

```

Assuming `Age` is declared as an integer (a reasonable assumption), if the user enters a character for the age, Ada raises an exception and terminates the program. More specifically, Ada raises the `Data_Error` exception because the data type of the user input is incompatible with the data type of the variable we're trying to read into. Let's look at the syntax for exception handlers in general, then add an exception handler here.

We first put the word `exception` to tell the compiler that we're including exception handlers. For each exception we want to handle, we put `when` followed by the exception name (such as `Data_Error`), then the statements we want to execute if the exception is raised. We can use `when others` to specify statements that should be executed if an exception is raised but it's not one of the exceptions listed by name (similar to `when others` for the case statement).

Exception Handler

```
exception
```

```
    when <exception name> => <executable statements>
```

```
    when <exception name> => <executable statements>
```

```
    . . .
```

```
    when others => <executable statements>
```

<exception name> - the name of an exception.

<executable statements> - one or more executable statements.

We often put the exception handler at the end of a procedure, where it will handle exceptions raised anywhere in the procedure. For our problem, however, we'd like to let the user re-enter an age, so this wouldn't quite work for us. We know we need to loop, exiting when we get a valid (correct data type) age, but we also need to add one more thing. Exception handlers work within a particular *block* (a portion of code surrounded by a *begin* and an *end*). We can insert a *begin* and an *end* to form a block wherever we want one in our code, and we can form as many blocks as we'd like. Here's how we do this for our problem:

```
-- loop
loop
    begin

        -- Prompt for and get age
        Put (Item => "Please enter age : ");
        Get (Item => Age);
        Skip_Line;

        -- exit from loop on numeric entry
        exit;

exception
    when Data_Error =>
        Put_Line (Item => "Input must be a number !!!");
        Skip_Line; -- need this because exception
                   -- raised before skip_line above
```

```
    end;  
end loop;
```

It turns out we can also define our own exceptions and then explicitly raise and handle them in our code, but those topics are beyond the scope of this book.

Bibliography

- [Ans95] ANSI/ISO/IEC-8652:1995, *Ada 95: The Language Reference Manual & Standard Libraries*, 1995.
- [Bar94] Barnes, J.G.P., *Programming in Ada Plus an Overview of Ada 9X*, Addison-Wesley Publishing Company, Massachusetts, 1994.
- [FK96] Feldman, Michael B., and Koffman, Elliot B., *Ada 95: Problem Solving and Program Design*, Addison-Wesley Publishing Company, Massachusetts, 1996.
- [Lom87] Lomuto, Nico, *Problem Solving Methods with Examples in Ada*, Prentice-Hall, New Jersey, 1987.
- [Mit84] Mitchell, William, *Prelude to Programming: Problem Solving and Algorithms*, Reston Publishing Co., Virginia, 1984.
- [Pol54] Polya, G., *How to Solve It*, Princeton University Press, New Jersey, 1954.
- [Wic79] Wickelgren, Wayne A., *Cognitive Psychology*, Prentice-Hall, New Jersey, 1979.
- [Woo94] Woods, Donald R., *Problem-Based Learning: How to Gain the Most from PBL*, Donald R. Woods, Waterdown, Ontario, 1994.

Index

A

Actual parameters, 77-81
 Ada.Float_Text_IO, 55
 Ada.Integer_Text_IO, 54
 Ada.Text_IO, 40, 57
 Algorithms, 1, 8, 11-12
 Arrays
 accessing elements of,
 140-141
 elements, 139-141
 indexes, 139, 143
 multi-dimensional, 143-144
 type definition, 138-140
 walking, 141-143
 Assignment statements, 52-53

B

Block, 241
 Boolean
 data type, 50
 expressions, 15
 I/O, 59-60
 operators, 50

C

Calling, procedures, 76-80
 Case statements, 98-100
 Character data type, 48
 Clauses
 use, 40
 with, 40
 Closing files, 167-168

Comments, 39-40
 Condition-controlled loops, 122
 Constants
 declarations, 47
 and variables, 41
 Control Flow Graphs (CFGs)
 defined, 13
 iteration, 17
 selection, 14-16
 sequential, 13
 Control structures
 combined, 18-19
 iteration, 16-18
 selection, 14-16
 sequential, 12-13
 Count-controlled loops, 118
 Creating files, 170-171

D

Data types
 Boolean, 50
 Character, 48
 defined, 46-47
 Float, 48
 Integer, 48
 String, 48-49
 Declarations
 Constant, 47
 Variable, 47
 Decomposition
 diagrams, 20-22
 Top-down, 20-22

Definitions, type
 array, 138-140
 enumeration type, 232
 record, 233
 string, 49

E

Else, 94-95
 Elsif, 96-98
 End_of_File, 168-170
 End_of_Line, 168-170
 Enumeration types, 230-232
 Exception handling, 240-242
 Expressions, Boolean, 15

F

Fields, of a record, 233-235
 Files
 closing, 167-168
 creating, 170-171
 modes, 165-166, 170
 opening, 165-166
 reading from, 166
 variables, 164-165
 writing to, 171
 Float data type, 48
 For loops, 118-122
 Formal parameters, 72-73
 Functions, 236-237

G

Get, 42, 52, 54, 55, 57
 Get_Line, 58

H

Handling, exceptions, 240-242

I

Identifiers, 40-41
 If statements
 one-alternative, 92-94
 two-alternative, 94-95
 multiple-alternative, 95-98
 nested, 98
 Indexes, array, 139, 143
 Infinite loops, 124
 Input
 Boolean, 59
 character, 57
 file, 164-170
 float, 55
 integer, 54
 string, 57-58
 Integer data type, 48
 Iteration control structure, 16-18

L

Local variables, 67, 75
 Loops
 basic, 125-126
 condition-controlled, 122
 count-controlled, 118
 for, 118-122
 infinite, 124
 nested, 144
 while, 122-125

M

Mistakes, common
 chapter 6, 64-66
 chapter 7, 90-91
 chapter 8, 116
 chapter 9, 136

chapter 10, 163
chapter 11, 188-190

Modes

file, 165-166, 170
parameter, 72

Multiple-alternative `if`, 95-98

N

Named association, 78-79

Nested `if` statements, 98

Nested loops, 144

`New_Line`, 42

O

One-alternative `if`, 92-94

Opening files, 165-166

Operators

Boolean, 50
relational, 93
table, 51

Output

Boolean, 59-60
character, 58
file, 170-171
float, 56
integer, 54-55
string, 58-59

Overloading, 237-240

P

Packages

bodies, 224
specifications, 223-224
with and use, 40

Parameters

actual, 77-81
formal, 72-73
modes, 72
passing, 77-79

Precedence, order of, 93-94

Problem solving, 4-10

Procedures

body, 224
calling, 76-80
specifications, 223
writing, 67-76

Prompting, 13, 42

`Put`, 42, 54-55, 56, 58-59

`Put_Line`, 59

R

Records, 233-236

`Rem`, 48

Reserved words, 39

S

Scientific notation, 56

Selection control structure, 14-16

Sequential control structure,
12-13

`Skip_Line`, 42

String data type, 48-49

Subtype, string, 49

Syntax boxes

Ada program syntax, 38
array element operations,
141

- array type definition and
variable declaration,
139
- assignment statements, 53
- basic loop syntax, 125
- case statement, 98
- closing files, 167
- creating files, 170
- enumeration type definition
syntax, 232
- exception handler, 241
- file variable declaration,
165
- for loop syntax, 119
- multiple-alternative if
statement, 96
- one-alternative if
statement, 93
- opening files, 165
- package body syntax, 225
- package specification
syntax, 224
- procedure calls, 78
- procedure headers, 72
- procedure syntax, 67
- record type definition and
variable declaration,
233
- two-alternative if
statement, 94
- variable and constant
declaration, 47
- while loop syntax, 123

T

Testing

- combinations, 31-34
- iteration, 29-31
- plans, 8, 25, 35
- selection, 26-29
- sequential, 24-26

Top-down decomposition, 20-22

Types

- array, 138-140
- Boolean, 50
- character, 48
- enumeration, 230-232
- file, 165
- float, 48
- integer, 48
- record, 233-234
- string, 48-49

U

- Use clause, 40

V

Variables

- and constants, 41
- declarations, 47
- giving a value, 52-53

W

- While loops, 122-125
- With clause, 40